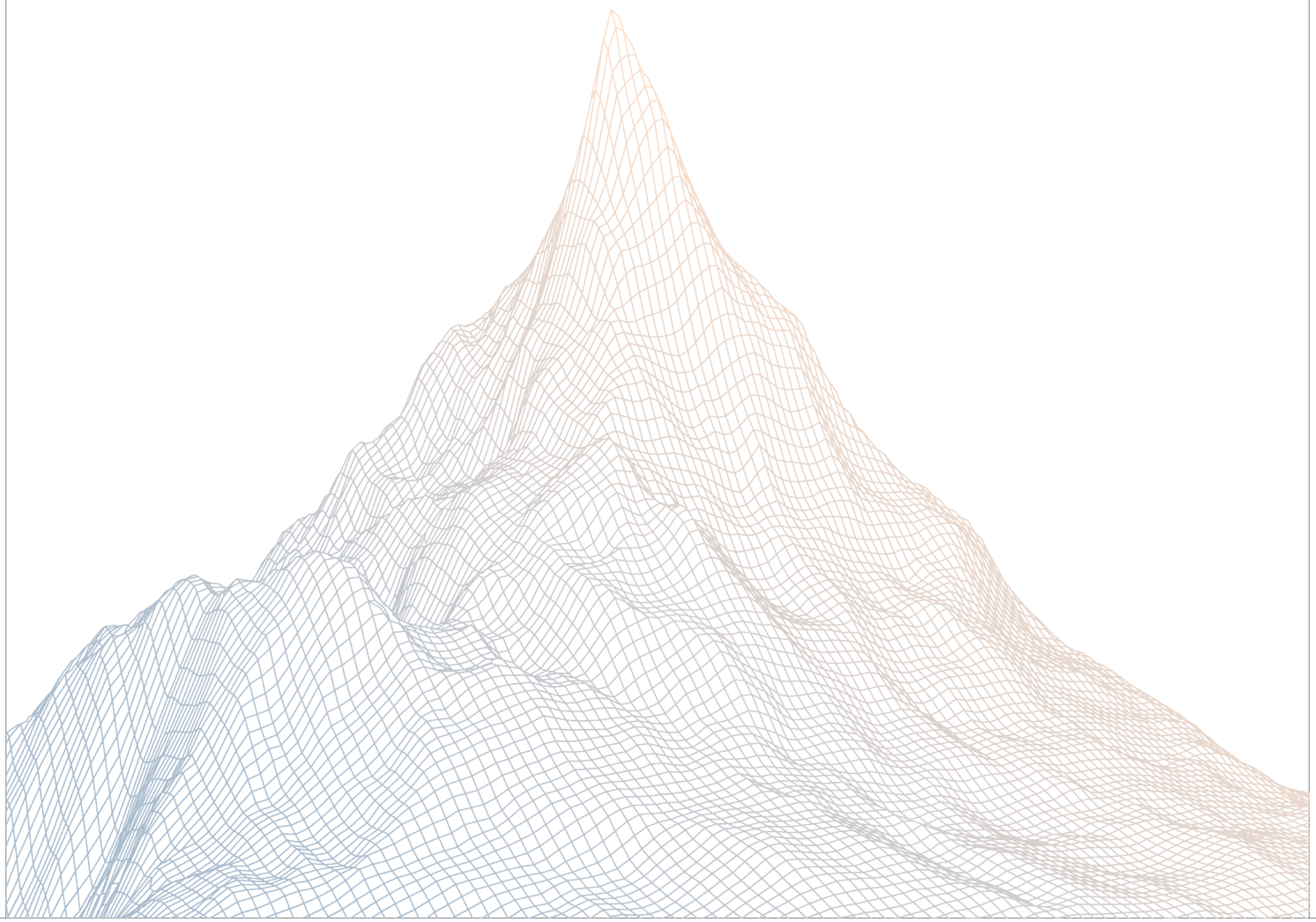
Zenith

Fluid

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: June 24th to July 31st, 2025

AUDITED BY: IIIIIIII000
kriko.eth
shaflow2

Contents

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 Jupiter

Jupiter Lend is shaping the future of finance on Solana by Leveraging Fluid's architecture, a two-layer architecture that separates liquidity to deliver greater 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
Commit Hash	`d2f9a01fe8f63a4cb9db1035e4b97cbc806a71ed`
Files	`programs/*`

2.3 Audit Timeline

June 24th, 2025	Audit start
July 31st, 2025	Audit end
August 14th, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	6
High Risk	3
Medium Risk	16
Low Risk	28
Informational	12
Total Issues	**65**

3

Findings Summary

ID	Description	Status
C-1	Incorrect bit masks break the branching algorithm	Resolved
C-2	Seed collision in vault accounts	Resolved
C-3	Tick 0 is incorrectly handled as "no tick exists"	Resolved
C-4	Attackers can use arbitrary TokenReserve accounts to undermine the protocol	Resolved
C-5	tick_id_data_after_operate account is missing a vault_id check	Resolved
C-6	Price calculation overflow can cause asset underpricing	Resolved
H-1	Transaction account count limits will break the vault	Acknowledged
H-2	Full liquidation did not clear the position debt in the fetch_latest_position function	Resolved
H-3	absorb function fails due to double account borrow blocking liquidations	Resolved
M-1	Meaning of liquidity.status is inverted	Resolved
M-2	The reward_rate may be released incorrectly.	Resolved
M-3	Potential overflow leading to DoS when calculating reward rate	Resolved
M-4	Wrong borrow rate may be used after calls to update_token_config()	Resolved
M-5	Some Token-2022 extensions may break the protocol	Acknowledged
M-6	Calls to update_user_borrow/supply_config() break accounting	Resolved
M-7	The vault's reserved tokens may be insufficient to cover the unclaimed tokens	Resolved
M-8	Wrong oracle prices used for liquidations	Resolved
M-9	calculate_new_token_exchange_price will revert if LRRM reward start time is in the future	Resolved

ID	Description	Status
M-10	Decreasing interest rates will DoS the system	Resolved
M-11	The withdrawal gap feature incorrectly reduces amount available for withdrawal by orders of magnitude	Resolved
M-12	Use of global counters in PDA derivations will allow DOS	Resolved
M-13	Overflow in debt calculation prevents position updates for large borrowers	Resolved
M-14	Positions above liquidation threshold after repayment may delay other liquidations	Resolved
M-15	Chain rollbacks may cause misconfiguration	Resolved
M-16	Stale prices may allow loans using worthless collateral	Resolved
L-1	The program initialization may be vulnerable to a front-running attack	Acknowledged
L-2	Using the deprecated transfer instruction may cause incompatibility with some tokens	Resolved
L-3	The f_token metadata will not be initialized.	Resolved
L-4	Compound interest deviation in borrow exchange price calculation	Acknowledged
L-5	No way to close and reclaim rent for some user-related accounts	Resolved
L-6	Incorrect behavior of checked_floor_div	Resolved
L-7	Users may be unable to withdraw dust amounts	Acknowledged
L-8	Some events may contain incorrect information	Resolved
L-9	Missing a check in init_lending to ensure that the mint and f_token_mint token programs are equal	Resolved
L-10	User supplies and borrows can't be paused independently	Resolved
L-11	Incorrect URI passed to Metaplex	Resolved

ID	Description	Status
L-12	Incorrect token_reserves_liquidity account for init_lending() ix may require redeployment	Resolved
L-13	The CLAIM token transfer_type does not support withdrawing and borrowing to different users.	Resolved
L-14	Missing default Pubkey checks	Resolved
L-15	Borrow/supply magnification may be applied to exchange price growth prior to vault creation	Resolved
L-16	The withdrawal_cap is not capped at 100%	Resolved
L-17	Vaults cannot work with mixed legacy/Token-2022 borrow/supply tokens	Resolved
L-18	No way to change a vault's authority	Acknowledged
L-19	Admin configuration updates can cause limit calculation discrepancies	Acknowledged
L-20	Use of pause_user() prior to setting user configuration breaks subsequent configuration	Resolved
L-21	get_ticks_from_remaining_accounts() uses the wrong error code when program ownership checks fail	Resolved
L-22	add_debt_to_tick() never executes the initialization code path	Resolved
L-23	The governance signer may lose auth or guardian roles	Resolved
L-24	Unrestricted lending program account in rewards transition	Resolved
L-25	Prior period rewards may be lost if LRRM.start_rewards() is called again	Resolved
L-26	Open TODOs	Resolved
L-27	Vector field sizes not validated when updating accounts	Resolved
L-28	Internal errors are not converted to external ones	Acknowledged
I-1	Typos	Resolved

ID	Description	Status
I-2	Dead code	Resolved
I-3	Accounts need not be marked as mutable	Resolved
I-4	Delayed reward distribution may temporarily prevent withdrawals	Acknowledged
I-5	The Operate instruction may be called with some accounts that are not used.	Resolved
I-6	Unused instruction accounts	Resolved
I-7	Unused account fields waste rent	Resolved
I-8	The init_token_reserve instruction does not check whether the token decimals are supported	Resolved
I-9	Misleading comments	Resolved
I-10	Unhandled SKIP token transfer_type	Acknowledged
I-11	There may not be enough user class entries to support future tokens	Acknowledged
I-12	Rewards calculation does not credit interest-related TVL growth	Resolved

4

Findings

4.1 Critical Risk

A total of 6 critical risk findings were identified.

[C-1] Incorrect bit masks break the branching algorithm

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- programs/vaults/src/module/user.rs
- programs/vaults/src/state/tick_has_debt.rs

Description:

In Rust, the bit shift operators have lower precedence than the arithmetic operators. This means that `1 << 4 - 1` results in `00001000` rather than `00001111`. The code that is attempting to create a bit mask like the latter, is using syntax that creates the former. This results in the bit mask not actually setting the lower bits, and instead sets only the indexed bit. This has the effect of clearing lower ticks in the `tick_has_debt` data, which causes the code that attempts to find the next pure tick, to return the incorrect result, breaking the branching algorithm.

Recommendations:

Add parentheses around the shift operation.

Fluid: Resolved with @56d82546457...

Zenith: Verified.

[C-2] Seed collision in vault accounts

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- vaults/src/state/context.rs

Description:

The seeds of the following account types in the `vault` program can collide with other accounts of the same type:

- `BranchData`
- `TickHasDebtArray`
- `TickData`
- `TickIdLiquidation`
- `UserPosition`
- `Mint` (in `InitPosition`)

We'll demonstrate the issue on the `BranchData` account, although the principle applies to all the mentioned account types.

```
#[account(
    init,
    payer = signer,
    space = 8 + BranchData::INIT_SPACE,
    seeds = [BRANCH_SEED, vault_id.to_string().as_bytes(),
    branch_id.to_string().as_bytes()],
    bump
)]
pub branch: AccountLoader<'info, BranchData>,
```

As we can see in the code, the seeds of the `BranchData` are built using two `u16` and `u32` parameters, which are first converted to String and then to bytes. This, however, means that `vault_id = 1` and `branch_id = 11` would be converted to `"1"` and `"11"` in the string form, hence `[49]` and `[49, 49]` in the bytes form, respectively. `vault_id = 11` and `branch_id = 1` would be converted to `"11"` and `"1"`, meaning `[49, 49]` and `[49]` in the bytes form.

This means that the seeds of two different `Branch` accounts would be the same, hence

producing the same PDA:

```
seeds 1: [[98, 114, 97, 110, 99, 104], [49], [49, 49]]
seeds 2: [[98, 114, 97, 110, 99, 104], [49, 49], [49]]
```

This would lead to some instructions not being callable due to accounts being already initialized. Additionally, this collision could allow users to access or modify accounts they shouldn't control, leading to potentially disastrous scenarios.

Recommendations:

Consider converting the seed parameters to little-endian bytes with the `to_le_bytes` method instead, and passing them to the seeds array using `as_slice`, which preserves the binary format of the integers, avoiding collisions:

```
#[account(
    init,
    payer = signer,
    space = 8 + BranchData::INIT_SPACE,
    seeds = [BRANCH_SEED, vault_id.to_string().as_bytes(), branch_id.to_
        string().as_bytes()],

    seeds = [BRANCH_SEED, vault_id.to_le_bytes().as_slice(), branch_id.to_le_
        bytes().as_slice()],
    bump
)]
pub branch: AccountLoader<'info, BranchData>,
```

For comparison, the same parameters used in the report would produce different seeds (hence different PDAs):

```
seeds 1: [[98, 114, 97, 110, 99, 104], [1, 0], [11, 0, 0, 0]]
seeds 2: [[98, 114, 97, 110, 99, 104], [11, 0], [1, 0, 0, 0]]
```

Note that little-endian encoding is recommended here as it is the standard byte order used throughout Solana for numerical data serialization.

Fluid: Resolved with @e617d03d457... and @3c50e0cb177...

Zenith: Verified.

[C-3] Tick 0 is incorrectly handled as "no tick exists"

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- vaults/src/module/user.rs
- structs.rs

Description:

The system incorrectly treats legitimate tick 0 positions as "no tick exists", causing liquidation failures and corruption of the vault's risk management system.

Consider a position with `1000000000000` collateral and `999999999999` debt. The ratio of this position is `281474976710655`, and the `get_tick_at_ratio` function would return `-1`, which means the actual tick increased by 1 would be `0`. If the exchange price is `1e18`, with supply exchange price = `1112777777777`, borrow exchange price = `1000000000000`, and collateral factor of 90%, the calculated tick comes to zero, meaning this is a valid and healthy position, and the topmost tick will be set to zero. This means that even if the positions in this vault become unhealthy, liquidation will become impossible due to the topmost tick being zero.

When the topmost tick is legitimately set to 0, all liquidation attempts become impossible due to the validation check in the `liquidate` function, which incorrectly treats `topmost_tick = 0` as "no tick exists":

```
if vault_state.topmost_tick == 0 {
    return Err(error!(ErrorCodes::VaultTopTickDoesNotExist));
}
```

Additionally, this creates a cascading corruption issue. When a legitimate tick 0 position is created, the comparison `if memory_vars.tick ⩾ top_tick` becomes true, setting `topmost_tick = 0`. On the next operation, `vault_state.get_top_tick()` returns `i32::MIN` because it converts stored 0 to `i32::MIN`. This means any subsequent position at tick -100, -50, etc. will satisfy `memory_vars.tick ⩾ i32::MIN` and incorrectly become the new "topmost" tick.

The issue stems from the codebase treating 0 as a sentinel value for "no tick exists" in the `get_top_tick()` function:

```
pub fn get_top_tick(&self) → i32 {
    if self.topmost_tick == 0 {
        i32::MIN
    } else {
        self.topmost_tick
    }
}
```

However, this behavior is incorrectly ported from the Solidity implementation, which correctly distinguishes between these states using bit encoding. In the Solidity version, tick = 0 is encoded as `(sign=1, absolute=0)` → bit pattern = `4` (decimal) → passes check, while "no tick exists" is encoded as `(sign=0, absolute=0)` (meaning negative zero) → bit pattern = `0` (decimal) → fails check.

Similarly, In `set_branch_data_in_memory`, if `connected_minima_tick` is 0, it is treated as an invalid tick. This is inappropriate.

```
pub fn set_branch_data_in_memory(&mut self, branch: &Branch) → Result<()> {
    ...
    self.minima_tick = self.data.connected_minima_tick;

    // if 0 that means it's a master branch
    if self.minima_tick == 0 {
        self.minima_tick = i32::MIN;
    }

    Ok(())
}
```

For example, suppose a liquidation occurs and the liquidation tick happens to be 0. The branch's tick would then be set to 0. If the collateral price later recovers, a user may open a position with a tick higher than `top_tick`, causing the branch to be created and connect to the previous branch with a tick of 0. As a result, the current branch's `connected_minima_tick` becomes 0.

If the collateral price drops again and another liquidation occurs, with the liquidation tick < 0, then after liquidating the `top_tick`, the connected branch will not be liquidated because its `minima_tick` is set to `i32::MIN`. However, the connected branch should in fact be liquidated, since it is at tick = 0, which is greater than the liquidation tick.

Recommendations:

Consider updating the validation logic to use `i32::MIN` as the "no tick exists" sentinel value:

```
if vault_state.topmost_tick == 0 {
if vault_state.topmost_tick == i32::MIN {
    return Err(error!(ErrorCodes::VaultTopTickDoesNotExist));
}
```

Additionally, ensure consistent state management by updating the functions in `vault_state.rs`:

```
pub fn reset_top_tick(&mut self) {
   self.topmost_tick = 0;
   self.topmost_tick = i32::MIN;
}
```

```
pub fn set_top_tick<'info>(

    // ... snippet

    if new_top_tick == i32::MIN {
        // Last user left the vault
       self.topmost_tick = 0;
       self.topmost_tick = i32::MIN;
        self.reset_branch_liquidated();
    }

    // ... snippet
}
```

```
pub fn get_top_tick(&self) -> i32 {
   if self.topmost_tick == 0 {
       i32::MIN
   } else {
       self.topmost_tick
   self.topmost_tick
   }
}
```

Lastly, the `init_vault_state` function should set the topmost tick to the `i32::MIN` value.

Fluid: Resolved with @8bd3005b457..., @ee28bc1d4b6..., @09e1c4aff71..., and @8ecdb86a39b...

Zenith: Verified.

[C-4] Attackers can use arbitrary TokenReserve accounts to undermine the protocol

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- vaults/src/state/context.rs
- vaults/src/module/user.rs

Description:

The `operate` function accepts arbitrary `TokenReserve` accounts for both supply and borrow tokens without validating that these accounts correspond to the correct tokens for the vault. This allows attackers to manipulate the supply exchange price used in collateral factor calculations by providing a `TokenReserve` account from a different, more valuable token.

The token reserve accounts are not validated against the vault's configured tokens in the `Operate` context:

```
#[account(mut)]
pub supply_token_reserves_liquidity: AccountLoader<'info, TokenReserve>,
#[account(mut)]
pub borrow_token_reserves_liquidity: AccountLoader<'info, TokenReserve>,
```

The `load_exchange_prices` function blindly trusts these user-provided accounts:

```
let supply_token_reserves = supply_token_reserves_account.load()?;
let borrow_token_reserves = borrow_token_reserves_account.load()?;
// ...
let (liq_supply_ex_price, _)
    = supply_token_reserves.calculate_exchange_prices()?;
let (_, liq_borrow_ex_price)
    = borrow_token_reserves.calculate_exchange_prices()?;
```

While these accounts are expected to be validated when calling `operate` on the Liquidity program, during borrow-only operations, only the borrow token reserve is checked by the liquidity program, leaving the supply token reserve unchecked. However, the vault still loads the exchange price from the unchecked supply token reserve for collateral factor

calculations.

An attacker can exploit this by providing a `TokenReserve` account from a different token as the `supply_token_reserves_liquidity`. If the vault's legitimate collateral `TokenReserve` has a supply exchange price of 1e12 and the attacker provides a `TokenReserve` from a different token with a supply exchange price of 1.5e12, this inflates the supply exchange price used in the collateral factor calculation, allowing the attacker to borrow 50% more than their collateral should support.

The collateral factor calculation uses `exchange_rate = exchange_rate.safe_mul(supply_ex_price)?.safe_div(borrow_ex_price)?;`. By substituting a higher `supply_ex_price` from a different token's reserve, the attacker artificially inflates the `exchange_rate`, which increases the `ratio_at_cf` threshold, allowing excessive borrowing.

This enables an attack where the attacker can over-borrow using inflated exchange prices, then liquidate the over-leveraged position (creating bad debt for the protocol), and repeat the process with new positions to drain protocol funds.

Similarly, in the `liquidate` instruction, the `TokenReserve` account does not undergo a mint matching check because the program expects this to be verified during the CPI call.

```
#[derive(Accounts)]
pub struct Liquidate<'info> {
    // ...
    // Here we will list all liquidity accounts, needed for CPI call to
    liquidity program
    // @dev no verification of accounts here, as they are already handled in
    liquidity program,
    // Hence loading them as AccountInfo
    // No need to verify liquidity program key here, as it will be verified
    in liquidity program CPI call
    #[account(mut)]
    pub supply_token_reserves_liquidity: AccountLoader<'info, TokenReserve>,
    #[account(mut)]
    pub borrow_token_reserves_liquidity: AccountLoader<'info, TokenReserve>,
    // ...
}
```

However, there is a special case in the `liquidate` instruction: a user can pass in `debt_amt = 0` to choose to settle only bad debt. In this case, the instruction exits immediately after settling the bad debt, and no CPI call occurs. Therefore, the correctness of the passed-in `TokenReserve` account cannot be guaranteed.

```
pub fn liquidate<'info>(
    ctx: Context<'_, '_, 'info, 'info, Liquidate<'info>>,
    debt_amt: u64,
```

```
    col_per_unit_debt: u64, // min collateral needed to receive per unit of
    debt paid back in 1e18
    absorb: bool,
    remaining_accounts_indices: Vec<u8>, // first index is sources, second
    is branches, third is ticks, fourth is tick has debt
) -> Result<(u128, u128)> {
    // ...
            // Call absorb function to handle bad debt above max limit
            // @dev passing vault_state as a mut reference, that means it
    will be updated inside the absorb function
            self::absorb(
                &mut vault_state,
                &tick_accounts,
                &tick_has_debt_accounts,
                &tick_has_debt_indices,
                &branch_accounts,
                &mut ctx.accounts.new_branch,
                memory_vars.max_tick,
            )?;

            if debt_amt == 0 {
                // If debt_amt was 0, we just wanted to absorb
                return Ok((0, 0));
            }
        }
    // ...
}
```

A malicious actor can choose to pass in `debt_amt = 0` along with a `borrow_token_reserves_liquidity` account whose mint does not match. If the `liq_borrow_ex_price` of this account is higher than the actual price, it will cause the calculated `vault_borrow_ex_price` to be overstated, which ultimately results in a lower calculated `max_tick`. This causes ticks that should not be classified as bad debt to be treated as bad debt and settled, leading to users' loss of funds.

Recommendations:

Add constraints to ensure `supply_token_reserves_liquidity` and `borrow_token_reserves_liquidity` correspond to the vault's configured supply and borrow tokens:

```
#[account(
    mut,
    constraint = supply_token_reserves_liquidity.load()?.mint
    == vault_config.load()?.supply_token
```

```
)]
pub supply_token_reserves_liquidity: AccountLoader<'info, TokenReserve>,

#[account(
    mut,
    constraint = borrow_token_reserves_liquidity.load()?.mint
    == vault_config.load()?.borrow_token
)]
pub borrow_token_reserves_liquidity: AccountLoader<'info, TokenReserve>,
```

Fluid: Resolved with @94a39a5913d... and @7c235c9f8bf...

Zenith: Verified.

[C-5] `tick_id_data_after_operate` account is missing a `vault_id` check

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- validate.rs

Description:

In the `Operate` instruction, the `tick_id_data_after_operate` account is missing a `vault_id` check in `verify_operate`.

```
pub fn verify_operate<'info>(ctx: &Context<'_, '_, 'info, 'info,
    Operate<'info>>) → Result<()> {
    // ...
    let tick_id_data = &ctx.accounts.tick_id_data;
    if tick_id_data.vault_id ≠ vault_state.vault_id {
        return Err(error!(ErrorCodes::VaultInvalidVaultId));
    }

    let new_branch = &ctx.accounts.new_branch.load()?;
    if new_branch.vault_id ≠ vault_state.vault_id {
        return Err(error!(ErrorCodes::VaultInvalidVaultId));
    }

    Ok(())
}
```

This allows a malicious actor to pass in a `tick_id_data_after_operate` account with a mismatched `vault_id`, causing the post tick liquidation history to be written into the `TickIdLiquidation` account of another vault.

This can lead to loss of funds for other users. Because the historical position update involves a `connection_factor` of 0, it results in a division-by-zero panic.

```
            let mut current_connection_factor:
    u128 = connection_factor.cast()?;
```

```
            // ...
            } else {
                // If branch is not merged, the main branch it's connected to
    then it'll have minima debt factor
                // position debt = debt * base branch minimaDebtFactor /
    connectionFactor
                let branch_min_debt_factor:
    u128 = branches[current_branch_idx]
                    .load()?
                    .get_branch_debt_factor()?;

                position_raw_debt = mul_div_normal(
                    position_raw_debt,
                    branch_min_debt_factor.cast()?,
@>                   current_connection_factor.cast()?,
                )?;
```

Recommendations:

Check whether the `vault_id` of the `tick_id_data_after_operate` account matches in the `verify_operate` function.

Fluid: Resolved with @f84a92a6c1f...

Zenith: Verified.

[C-6] Price calculation overflow can cause asset underpricing

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- programs/oracle/src/helper.rs

Description:

The exchange rate of assets is in 18 decimals precision, since the `rate` is initially set to 1e18 (see here), meaning that `get_exchange_rate_for_hop` must return prices in 18 decimals precision.

The Pyth oracle, for example, returns price along with the `exponent` value, which is the decimal precision for that price feed. We assume that this is handled with proper `multiplier` and `divisor` values, and the `rate` is properly converted to 18 decimals here.

However, if we consider a BTC/USD price feed from Pyth, the returned price at the time of writing is `11833017161230`, with exponent = `-8`, meaning 8 decimals. This price should therefore be multiplied by `1e10` in order to get it to 18 decimals, yielding `118330171612300000000000`.

This, however, means that the following calculation:

```
rate = rate
    .saturating_mul(current_hop_rate)
    .saturating_div(10u128.pow(RATE_OUTPUT_DECIMALS));
```

would overflow, but since `saturating_mul` is used, it would not fail but saturate at `u128::MAX`. The division would then yield `340282366920938463463` in 18 decimals, which comes to `340.282366920938463463` USD, which would be a disastrous discount.

Recommendations:

Consider either decreasing the value of `RATE_OUTPUT_DECIMALS` or implementing the logic utilizing the `U256` type available in the `uint` crate.

Fluid: Resolved with @fb6fa428125...

Zenith: Verified.

4.2 High Risk

A total of 3 high risk findings were identified.

[H-1] Transaction account count limits will break the vault

SEVERITY: High	IMPACT: High
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- programs/vaults/src/state/context.rs

Description:

Solana transactions are limited to 1232 bytes, which is the amount of data that can fit in a single network packet. When taking into account the size of headers, signatures, and addresses, this means legacy transactions can only fit about 34 addresses in a transaction. When using Address Lookup Tables, this limit expands significantly, but has a hard maximum of 128 addresses. If a transaction requires more than this number of accounts, then the transaction will not be able to be created.

One area of the code that runs into this limitation is user `operate()` instructions. The `Operate` Accounts struct by itself lists 34 accounts, leaving 94 accounts available for provision in `remaining_accounts`. Two of these are consumed by the two oracle sources, and at least one account is consumed for a `tick_has_debt_accounts` account, leaving 91 for `BranchData` accounts. With the way that branches are created, it is likely for the accounts limitation to be hit. Consider the case of a range-bound market where users A, B, and C create positions, and then their branch is partially liquidated. Next, the market goes back up, and dip buyers create new positions at the top-most tick with positions which creates a new branch, and then some of them are liquidated. The market goes back up, new dip buyers take out new loans, and then are partially liquidated when the market goes back down. Each of these dip-buying-then-liquidation cycles creates a new branch that must be passed to user A, B, and C's `operate()` instructions in order for them to modify their positions. If there are enough cycles, those users will be unable to modify their positions, and their positions and collateral will be stuck permanently (until they're liquidated or absorbed).

The `liquidate()` instruction has a similar issue in that if there are a lot of branches that a large liquidation wants to cover, that transaction will require many branch data, tickdata, and has debt accounts, which will require them to break up their liquidations into multiple

smaller liquidations, rather than one large one.

Recommendations:

Design a multi-transaction version of `operate()` that allows processing, via a new mechanism, to be done in stages in different transactions, and ensure that the `liquidate()` limitation is documented in code comments, and that the SDK properly calculates the correct number of accounts to include in order to stay under the limit.

Fluid: Acknowledged. From our observations, branch depth rarely exceeds 5 in practice, which is what we've seen on Ethereum as well. For the operate instruction, we use Address Lookup Tables, giving us an effective limit of ~64 accounts enough to handle roughly 20—25 branches well above what we expect.

For liquidate, partial liquidation is already supported via the debt_amount parameter. We're also considering adding a branch-based restriction similar to debt_amount so that liquidations always go through, regardless of branch count.

In the meantime, this can be managed at the SDK level by calculating the appropriate debt_amount based on the passed branches or ticks to ensure partial liquidation when needed.

If needed in the future, we can also implement a multi-instruction approach to work around account limits entirely.

[H-2] Full liquidation did not clear the position debt in the `fetch_latest_position` function

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

Target

- structs.rs

Description:

When operating on a position, if a position was previously opened and its tick has been liquidated, the `fetch_latest_position` function needs to be called to retrieve the updated state after liquidation. However, during the tick is full liquidation, the program only sets `position_tick` to `i32::MIN` but does not reset `position_raw_debt` to zero.

```
pub fn fetch_latest_position(...) -> Result<usize> {
    //...
    let initial_position_raw_debt: u64 = self.debt_raw.cast()?;
    // ...
    let mut position_raw_debt: u64 = initial_position_raw_debt;
    let mut position_raw_col: u128 = 0;

    if is_fully_liquidated {
        position_tick = i32::MIN;
    } else {
        ...
    }
    self.tick = position_tick;
    self.debt_raw = position_raw_debt.cast()?;
    self.col_raw = position_raw_col;

    Ok(current_branch_idx)
}
```

This can lead to the user opening more debt than intended. For example, if User 1's position held 100 debt and was fully liquidated at some point, and now the user attempts to open a new position with 100 debt, the old 100 debt is not cleared from `memory_vars`, potentially resulting in 200 debt being unintentionally opened.

Recommendations:

When `is_fully_liquidated` is true, also set `position_raw_debt` to zero.

Fluid: Resolved with @b19b3dbc472...

Zenith: Verified.

[H-3] `absorb` function fails due to double account borrow blocking liquidations

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

Target

- programs/vaults/src/module/user.rs

Description:

The `absorb` function may fail in certain cases due to borrowing the same `BranchData` account twice. The function first loads a `BranchData` account (or here), then attempts to load the same account mutably within a loop. Since `branch_data.id` remains unchanged, the second borrow fails. If the current branch has not yet been liquidated, and the first historical branch connected to the current branch has become bad debt due to price fluctuations and needs to be absorbed, the above issue will occur.

The `absorb` function is triggered when `topmost_tick > max_tick`. When `absorb` fails, the entire liquidation system becomes blocked, preventing both bad debt absorption and liquidation of borderline positions.

Recommendations:

Restructure the function to avoid double borrowing by scoping the initial `BranchData` account load to extract all necessary data before attempting to borrow the account mutably in the processing loop.

Fluid: Resolved with @b70c44d2c0d...

Zenith: Verified.

4.3 Medium Risk

A total of 16 medium risk findings were identified.

[M-1] Meaning of `liquidity.status` is inverted

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

Target

- programs/liquidity/src/module/admin.rs

Description:

The `pre_operate()`, `operate()`, and `claim()` instructions each have the following constraint:

```
#[account(constraint = liquidity.status == true @
    ErrorCodes::ProtocolLockdown)]
pub liquidity: Box<Account<'info, Liquidity>>,
```

The constraint requires that the status is `true`, but elsewhere in the code and comments `true` is interpreted as the "paused" state. Since the initialization code sets the initial state to true, the instructions above are actually *un*paused by default. Attempts by auth roles to pause user operations will fail, requiring debugging, before determining that the input arg must be flipped instead.

Recommendations:

Change the constraints to use $\neq$ instead.

Fluid: Resolved with @e2ad489fabc...

Zenith: Verified.

[M-2] The `reward_rate` may be released incorrectly.

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

Target

- helpers.rs

Description:

When the current time exceeds `rewards_ended`, the program does not specially handle the final reward distribution up to the `rewards_ended` time. This may lead to incorrect reward allocation.

```
fn calculate_new_token_exchange_price(
    new_liquidity_exchange_price: u64,
    lending: &Account<Lending>,
    current_rate_model: &Account<LendingRewardsRateModel>,
    f_token_total_supply: u64,
) -> Result<(u64, bool)> {
    ...
    let (mut rewards_rate, rewards_ended, rewards_start_time)
    = current_rate_model.get_rate(
        old_token_exchange_price
            .cast::<u128>()?
            .safe_mul(f_token_total_supply.cast()?)?
            .safe_div(EXCHANGE_PRICES_PRECISION.cast()?)?
            .cast()?,
    )?;

    if rewards_rate > MAX_REWARDS_RATE.cast()? || rewards_ended {
        // rewardsRate is capped, if it is bigger > MAX_REWARDS_RATE, then
    the rewardsRateModel
        // is configured wrongly (which should not be possible). Setting
    rewards to 0 in that case here.
        rewards_rate = 0;
    }
```

For example: If the current reward distribution period is from time 100 to 200 and the `rewards_rate` is `rate1`, suppose the last system interaction occurs at time 190, and the next

one happens at time 210 — which is after `rewards_ended`. At that point, `rewards_rate` will immediately return 0, and the rewards between 190 and 200 will remain undistributed.

If there is another reward distribution queue starting after time 200 with `rewards_rate = rate2`, then only the rewards from 200 to 210 will be properly distributed. The rewards from 190 to 200 will still be lost.

Recommendations:

It is recommended to specifically handle the reward distribution for the period between the last update time and the `rewards_ended` time.

Fluid: Resolved with @b2ed0badecb...

Zenith: Verified.

[M-3] Potential overflow leading to DoS when calculating reward rate

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

Target

- lendingRewardRateModel/src/state/state.rs

Description:

When the new token exchange price is calculated in this helper function, the reward information is retrieved from the current rate model using the `get_rate` function.

When the current time is after the reward end time, the mathematical operations can overflow during the final rate calculation:

```
let yearly_reward = next_reward_amount
    .safe_mul(SECONDS_PER_YEAR)?
    .safe_div(next_duration)?;
let rate = yearly_reward
    .safe_mul(100_000_000_000_000)? // 1e14
    .safe_div(total_assets)?;
```

Since all of these values are of type `u64`, the overflow would occur if either `next_reward_amount` is greater than `584942417355`, or if the calculated `yearly_reward` is greater than `184467`.

Since this calculation is part of multiple `lending` operations, including `preview_mint`, `preview_redeem`, `execute_withdraw`, and `execute_deposit`, these would revert due to the overflow until the reward rate model's end time is updated or sufficient time passes beyond the calculated `new_end_time` threshold.

Recommendations:

Consider casting these values to `u128`:

```
let yearly_reward = next_reward_amount
   .cast::<u128>()?
```

```
    .safe_mul(SECONDS_PER_YEAR)?
    .safe_mul(SECONDS_PER_YEAR.cast()?)?
    .safe_div(next_duration)?;
    .safe_div(next_duration.cast()?)?;
let rate = yearly_reward
    .safe_mul(100_000_000_000_000)? // 1e14
    .safe_div(total_assets)?;
    .safe_div(total_assets.cast()?)?;
```

Fluid: Resolved with @b2ed0badecb...

Zenith: Verified.

[M-4] Wrong borrow rate may be used after calls to `update_token_config()`

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

Target

- programs/liquidity/src/module/admin.rs

Description:

If there are already live prices, the `update_token_config()` instruction calculates and stores the updated exchange prices after the configuration update. It also updates the `last_update_timestamp` but never updates the `last_utilization` or the `borrow_rate`, which are directly used to calculate the *next* interest owed/paid. It is possible that interest growth prior to the instruction, or the `fee_on_interest` change, would have otherwise pushed the utilization past the rate model's kink, and would have either required a much higher or much lower borrow rate than occurs due to this bug.

The solidity code has the same issue.

Recommendations:

Calculate and store the updated utilization and borrow rate as is done for the `operate()` instruction, or call `update_exchange_prices_and_rates()` after updating storage.

Fluid: Resolved with @9a5a9e6ad77... and @630b78c05ba...

Zenith: Verified

[M-5] Some Token-2022 extensions may break the protocol

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

Target

- programs/liquidity/src/module/user.rs
- programs/liquidity/src/state/context.rs
- programs/liquidity/src/module/admin.rs
- programs/liquidity/src/module/user.rs

Description:

Some token-2022 extensions may cause problems with the protocol:

- Permanent Delegate - The permanent delegate will always be authorized to transfer funds out of the vault and any claim account.
- Pausable - This extension will prevent all users from withdrawing funds for mints with this extension, since the liquidity program uses the deprecated `transfer()` function, which disallows this extension.
- Transfer Fees - In addition to breaking claims due to it not being supported for `transfer()`, mints with this feature will break the claim process since the user claim balance won't match the amount actually transferred into the claim account.
- Transfer Hooks - The deprecated `transfer()` function disallows this extension, which will break claiming.
- Default Account State - Claim accounts may be frozen, with no way for the `liquidity` PDA to unfreeze them.
- Mint Close Authority - A token with this extension may change its decimals by closing an re-opening with a new decimal amount. This may occur between when the token rate is first set, and when the token is first supplied, potentially violating the maximum unit amount constraints. This extension would also allow creating a new mint with a permanent delegate.

A related issue is that even normal SPL tokens have a freeze authority who can use that authority to prevent tokens from being transferred to/from any user's address. This may cause liquidations to fail or may prevent borrowers from repaying loans.

Recommendations:

Disallow tokens that have extensions enabled, and decide on a strategy for handling frozen accounts and token pauses.

Fluid: Made changes to exclude non-whitelisted extensions, but USDG requires support for some of the extensions listed above. Resolved with @dd1e564cfa4... and @PR-62.

Zenith: Fluid acknowledges that the dangerous extensions still are allowed in order to support USDG, and that later updates to the transfer fee, hooks, or default state may cause issues down the road.

[M-6] Calls to `update_user_borrow/supply_config()` break accounting

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

Target

- programs/liquidity/src/module/admin.rs

Description:

Unlike the solidity code, the `update_user_borrow/supply_config()` functions use `calculate_exchange_prices()` rather than calling `update_exchange_price()` which would also update the `last_update_timestamp`. This means that when the update functions are called, their internal calls to `update_exchange_prices_and_rates()` calculate fresh prices on top of what was fetched from `calculate_exchange_prices()`, resulting in a re-application of the expected interest since the last update.

Recommendations:

Use `update_exchange_price()` in both functions, rather than calling `calculate_exchange_prices()`

Fluid: Resolved with @9ea104b9dd6...

Zenith: Verified.

[M-7] The vault's reserved tokens may be insufficient to cover the unclaimed tokens

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

Target

- user.rs

Description:

If the withdrawal or borrow uses a Claim method, the user needs to manually call the `claim` instruction to receive their tokens. During this period, since the tokens remain in the vault, the corresponding amount must be recorded to prevent it from being borrowed or withdrawn. Otherwise, it could result in the user's claim failing due to insufficient funds.

In the `operate` instruction, there is an Inappropriate check: when `withdraw_to` ≠ `borrow_to`, the system processes the `withdraw` and `borrow` transfers sequentially. If the `Claim` transfer type is used in this case, the `borrow_transfer` might use an outdated `last_stored_claim_amount`, because `total_claim_amount` was already increased during the earlier `withdraw_transfer`.

```
pub fn operate(
    ctx: Context<Operate>,
    supply_amount: i128, // Used max available in rust i128
    borrow_amount: i128,
    withdraw_to: Pubkey,
    borrow_to: Pubkey,
    mint: Pubkey,
    transfer_type: TransferType,
) → Result<(u64, u64)> {
    ...
        } else {
            if withdraw_transfer_amount > 0 {
                let claim_amount = handle_transfer_or_claim(
                    &transfer_type,
                    withdraw_to,
                    &mut ctx.accounts.claim_account,
                    last_stored_claim_amount,
                    TokenTransferParams {
```

```
                    source: ctx.accounts.vault.to_account_info(),
                    destination:
ctx.accounts.withdraw_to_account.to_account_info(),
                    authority:
ctx.accounts.liquidity.to_account_info(), // the liqudity PDA owns the
authority to transfer the tokens
                    amount: withdraw_transfer_amount,
                    token_program:
ctx.accounts.token_program.to_account_info(),
                    signer_seeds: Some(&[&liquidity_seeds]),
                    mint: None,
                },
            )?;

            total_claim_amount =
total_claim_amount.safe_add(claim_amount)?;
        }

        if borrow_transfer_amount > 0 {
            let claim_amount = handle_transfer_or_claim(
                &transfer_type,
                borrow_to,
                &mut ctx.accounts.claim_account,
                last_stored_claim_amount,
                TokenTransferParams {
                    source: ctx.accounts.vault.to_account_info(),
                    destination:
ctx.accounts.borrow_to_account.to_account_info(),
                    authority:
ctx.accounts.liquidity.to_account_info(), // the liqudity PDA owns the
authority to transfer the tokens
                    amount: borrow_transfer_amount,
                    token_program:
ctx.accounts.token_program.to_account_info(),
                    signer_seeds: Some(&[&liquidity_seeds]),
                    mint: None,
                },
            )?;

            total_claim_amount =
total_claim_amount.safe_add(claim_amount)?;
        }
    }

    token_reserve.add_claim_amount(total_claim_amount)?;
}
```

For example:

1. The vault currently holds 3000 tokens, with `total_claim_amount = 2000`.
2. A request is made for `withdraw = 1000` and `borrow = 1000`, where `withdraw_to ≠ borrow_to`, and the `transfer_type` is `Claim`.
3. After executing `withdraw_transfer`, `total_claim_amount` should increase to 3000. At this point, no further tokens should be allowed to be withdrawn. However, since the outdated `last_stored_claim_amount = 2000` is passed into `borrow_transfer`, the subsequent `borrow_transfer` still passes the checks.
4. After both `borrow_to` and `withdraw_to` successfully call `claim`, the vault ends up with only 1000 tokens, while `total_claim_amount = 2000`. This may cause future user claims to fail due to insufficient funds in the vault.

Recommendations:

It is recommended to update the outdated `last_stored_claim_amount` before executing `borrow_transfer`.

Fluid: Resolved with @4625f94b131...

Zenith: Verified

[M-8] Wrong oracle prices used for liquidations

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

Target

- programs/vaults/src/invokes/oracle.rs

Description:

The oracle program interface supports two separate price-fetching variants used by the vault - `get_exchange_rate_operate()` and `get_exchange_rate_liquidate()`, where the later has extra limitations on the price returned, e.g. in order to prevent forced liquidation attacks. During liquidations, the vault calls its `get_exchange_rate_liquidate()` function, but this function ends up internally calling the `get_exchange_rate_operate()` instruction instead of the `get_exchange_rate_liquidate()` function, which means the vault program is vulnerable to forced liquidation attacks.

Recommendations:

Invoke the correct oracle CPI for liquidations.

Fluid: Resolved with @50187fee83d... and @0d5085d552a...

Zenith: Verified

[M-9] `calculate_new_token_exchange_price` will revert if LRRM reward start time is in the future

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

Target

- programs/lending/src/utils/helpers.rs

Description:

The `rewards_rate` from the Lending Rewards Rate Model is one of the factors when calculating the new token exchange price. The newly calculated price is inflated by the current rate from the LRRM, based on time elapsed since the last update until the current time. If the last update occurred before the current rewards start time, the `last_update_timestamp` is set to the `rewards_start_time` as we can see here. In the LRRM implementation, we can see that if the reward start time is in the future, hence **greater than the current time**, the function will return the start time of the rewards (see here).

This, however, means that later the function will try to subtract this future timestamp from the current timestamp as we can see here:

```
let total_return_from_rewards: u128 = rewards_rate
    .cast::<u128>()?
    .safe_mul(curr_timestamp.safe_sub(last_update_timestamp.cast()?)?)?
    .safe_div(SECONDS_PER_YEAR)?;
```

Note that this is the same implementation as in Solidity, where the implementation underflows since it is a part of an `unchecked` block, but it has no impact since the `reward_rate` in this case is zero. In the Rust implementation, the `curr_timestamp.safe_sub(last_update_timestamp.cast()?)` calculation will return an `Err` variant, hence DoSing the calls that rely on calculating the new token exchange price (which is one of the most essential functions in the protocol).

Recommendations:

Consider using `saturating_sub` instead of `safe_sub`:

```
let total_return_from_rewards: u128 = rewards_rate
    .cast::<u128>()?
   .safe_mul(curr_timestamp.safe_sub(last_update_timestamp.cast()?)?)?
   .safe_mul(curr_timestamp.saturating_sub(last_update_timestamp.cast()?))?
    .safe_div(SECONDS_PER_YEAR)?;
```

Fluid: Resolved with @b2ed0badecb...

Zenith: Verified.

[M-10] Decreasing interest rates will DoS the system

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

Target

- liquidity/src/state/rate_model.rs

Description:

As per inline comments here and here, the `declining rate is supported before kink. kink to max must be increasing.`. This means that the rate model can be set so that `rate_at_zero` is greater than `rate_at_kink1`, and it is an expected state. However, rate models set like this would not work because of underflow in the `get_rate` function.

The very first calculation is:

```
let num: i128 = y2.safe_sub(y1)?.safe_mul(TWELVE_DECIMALS)?.cast()?;
```

It represents the maximum rate minus the minimum rate. If the maximum rate is lower than the minimum (declining interest rate), the `safe_sub` would return an error, hence reverting. The result of this would be DoS of actions that would leave the utilization in the non-functioning decreasing rate area of the interest rate curve. The utilization would need to get back to the increasing rate area in that case for it to work.

Recommendations:

Consider casting `y2` and `y1` to `i128` before the subtraction:

```
let num: i128 = y2.safe_sub(y1)?.safe_mul(TWELVE_DECIMALS)?.cast()?;
let num: i128 = y2
        .cast::<i128>()?
        .safe_sub(y1.cast()?)?
        .safe_mul(TWELVE_DECIMALS.cast()?)?
        .cast()?;
```

Fluid: Resolved with @300b5ec7361...

Zenith: Verified

[M-11] The withdrawal gap feature incorrectly reduces amount available for withdrawal by orders of magnitude

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

Target

- programs/vaults/src/module/user.rs
- programs/vaults/src/utils/operate.rs

Description:

If the supply token being used has fewer than nine decimals, the amount of collateral deposited is scaled up to nine decimals. Later, this scaled up amount is passed to the `check_if_withdrawal_safe_for_withdrawal_gap()` function where it is divided by the `liquidity_ex_price`, which is an un-scaled amount. This means that the resulting amount being requested is over-estimated by the function, by an order of magnitude for every decimal the token is below nine. When the withdrawal gap feature is enabled, this will likely cause the withdrawal request to be rejected, because amounts that should have fit below the limit are treated as though they are much larger.

Recommendations:

Save the pre-scaled amount, and use it in the call to `check_if_withdrawal_safe_for_withdrawal_gap()`.

Fluid: Resolved with @e54760bf89a...

Zenith: Verified.

[M-12] Use of global counters in PDA derivations will allow DOS

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

Target

- programs/vaults/src/utils/validate.rs
- programs/vaults/src/state/context.rs

Description:

The creation of a new position requires that the caller provides the correct next position's ID, which is required to be equal to the count of the current number of positions. An attacker can cheaply DOS the creation of new positions by continuously creating random amounts of new positions in a transaction in every block with high priority fees, making it impossible for legitimate users of the protocol to anticipate the correct new position count.

The `init_tick_id_liquidation()` instruction also uses a global counter and therefore limits the speed with which users can modify their positions after a liquidation. The function requires that the new ID matches the `TickData`'s current `total_ids`, and this means new `TickIdLiquidations` can't be created until prior ones have been filled.

Recommendations:

For positions, allow random position IDs to be passed to the `init_position()` instruction. The `TokenMetadata` extension's `additional_metadata` field can be used on the NFT's mint in order to store the position's Pubkey, and the update of the total positions counter can be done the first time position is used in the `operate()` instruction.

For liquidations, allow the creation of multiple `TickIdLiquidations` at a time, so that they can be pre-inited prior to them being needed. It may also be worth while to increase the number of entries per `tick_map` in the `TickIdLiquidation` PDA, so that new `TickIdLiquidations` have to be created less frequently.

Fluid: For new position creation, we don't see a strong DOS risk currently, users can always retry later or we can patch via upgrade if needed. However, we agree that for position modification post liquidations this check is important, and we've implemented the proposed change by removing the check from `InitTickIdLiquidation` with @13b7d96b16d...

Zenith: Position creation DOS is acknowledged, and the liquidation DOS fix is verified.

[M-13] Overflow in debt calculation prevents position updates for large borrowers

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

Target

- programs/vaults/src/state/structs.rs

Description:

Users with an existing debt position can experience a denial of service due to an overflow in the `fetch_latest_position` function. If the user was liquidated before but retains more than 1% of the debt, the debt is decreased by 1% (see here). However, the `position_raw_debt` variable was declared as `u64` (see here), meaning that on values of `1844858893260282` and greater, it will overflow and panic. Since debt is standardized to 9 decimals, this would represent at least `1_844_858` units of debt, which in USDC, for example, can mean ~1.8 million USD. Since the function would not work, the user would not be able to update their position, potentially facing more damage.

Recommendations:

Consider casting the value to `u128` before multiplying:

```
position_raw_debt = position_raw_debt.safe_mul(9999)?.safe_div(10000)?;
position_raw_debt = position_raw_debt
    .cast::<u128>()?
    .safe_mul(9999)?
    .safe_div(10000)?
    .cast()?;
```

Fluid: Resolved with @819081bcac...

Zenith: Verified.

[M-14] Positions above liquidation threshold after repayment may delay other liquidations

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

Target

- programs/vaults/src/module/user.rs

Description:

Users are allowed to repay part of their debts without oracles being consulted, as long as the position results in a less risky tick with less net borrowing. While these sorts of operations make the position itself less risky, on Solana with the way the protocol's accounts are organized, such operations can introduce more risk into the system. Since any update to an existing position results in a completely new position with potentially new tick and "has debt" accounts created, this means that any user wishing to liquidate such a position will have to had provided these accounts in their liquidation instruction. Since most liquidations will involve the use of an Address Lookup Table which must have been 'activated' (its creation block must appear in the slot hashes sysvar) prior to its use, any changes to the accounts required for a liquidation ix will require a new ALT and 1-2 blocks of delay before the instruction can be executed. It is possible that a user is past the liquidation threshold and, repeatedly over multiple blocks, repays the minimum amount of debt, which ends up keeping them in the liquidation range over multiple blocks, and prevents the legitimate liquidation of it or other positions. Eventually, if the market moves quickly enough, positions may move to the absorption range and become uneconomical for liquidation.

Recommendations:

Do not let modified positions be added to ticks that are above the liquidation threshold according to the oracle price.

Fluid: Resolved with @50445ef3c98...

Zenith: Verified.

[M-15] Chain rollbacks may cause misconfiguration

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

Target

- programs/oracle/src/helper.rs

Description:

When there is a network outage the Solana chain is often rolled back to a slot in the past with a more consistent state. This means that if there have been configuration changes after that slot, the changes will be rolled back too. One place that is particularly vulnerable to this is the update of vault oracles. The Pyth oracles in use by the protocol use their own `multiplier` and `divisor` fields in order to ensure that the price returned has the correct decimals. Since the Pyth update's exponent is never read, the multiplier and divisor must necessarily account for the exponent. If the Pyth feed needs to change its exponent, then the Fluid protocol will need to update the vault to use a new oracle with new multipliers and divisors. If this change of oracles is rolled back, then users interacting with the vault prior to its re-application will be able to use prices with the wrong number of decimals, allowing them to liquidate positions that otherwise shouldn't be liquidated. Other configuration parameters such as the collateral factor, paused/frozen state, rate parameters, and fees may also have negative consequences when they're rolled back.

Recommendations:

Apply the Pyth exponent separately from the multiplier and divisor, and in all programs add account fields and instructions for tracking and updating the last restart slot, and prevent new user operations if it doesn't match the `LastRestartSlot` sysvar.

Fluid: Resolved with @5bd4a17f1a3.... While a positive exponent is theoretically possible, it has never occurred in practice. Additionally, we always validate the exponent before setting the feed, so if ever needed can be updated later. The other aspects of the finding relating to the last restart slot are acknowledged.

Zenith: Verified that the fix solves for the case of negative exponents changing. Positive exponents, and other configuration rollbacks are acknowledged.

[M-16] Stale prices may allow loans using worthless collateral

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

Target

- programs/oracle/src/modules/pyth.rs

Description:

On Solana, the Pyth network provides oracle updates every 60 seconds or whenever there is a deviation of 0.5% or more, for its sponsored feeds. For the non-sponsored feeds, there are no specific guarantees. The only caveat for the sponsored feeds is that if there is a network outage (which, in the past, has lasted for as many as 19 hours), the Pyth network may be unable to push updates until the outage is resolved. Since the Fluid protocol is reading the raw price update account without checking for staleness, when the network comes back an attacker can use flash loans to take out large loans using the outdated price and potentially worthless collateral, leaving the protocol with bad debt.

Recommendations:

Add price staleness checks such as the ones provided by the Pyth `get_price_no_older_than()` function.

Fluid: Resolved with @08a8d061673..., @b10e51a0ac7..., and @9ea1a950d34...

Zenith: Verified.

4.4 Low Risk

A total of 28 low risk findings were identified.

[L-1] The program initialization may be vulnerable to a front-running attack

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

Target

- context.rs

Description:

In the protocol, the initialization of privileged addresses is not protected by access control. If the project team does not perform the program deployment and initialization within the same transaction, a malicious actor could front-run the initialization, rendering the program unusable.

```
#[derive(Accounts)]
pub struct InitLendingAdmin<'info> {
    #[account(mut)]
    pub authority: Signer<'info>,

    #[account(
        init,
        payer = authority,
        space = 8 + LendingAdmin::INIT_SPACE,
        seeds = [LENDING_ADMIN_SEED],
        bump,
    )]
    pub lending_admin: Account<'info, LendingAdmin>,

    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
```

```rust
pub struct InitLendingRewardsAdmin<'info> {
    #[account(mut)]
    pub authority: Signer<'info>,

    #[account(
        init,
        payer = authority,
        space = 8 + LendingRewardsAdmin::INIT_SPACE,
        seeds = [b"lending_rewards_admin"],
        bump,
    )]
    pub lending_rewards_admin: Account<'info, LendingRewardsAdmin>,

    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct InitLiquidity<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,

    #[account(
        init,
        seeds = [LIQUIDITY_SEED],
        payer = signer,
        space = 8 + Liquidity::INIT_SPACE,
        bump
    )]
    pub liquidity: Account<'info, Liquidity>,

    #[account(
        init,
        seeds = [AUTH_LIST_SEED],
        payer = signer,
        space = 8 + AuthorizationList::INIT_SPACE,
        bump
    )]
    pub auth_list: Account<'info, AuthorizationList>,

    pub system_program: Program<'info, System>,
}
```

Recommendations:

It is recommended to add access control to these functions, restricting them so that only specific hardcoded addresses are allowed to call them.

Fluid: Acknowledged

[L-2] Using the deprecated `transfer` instruction may cause incompatibility with some tokens

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

Target

- spl.rs

Description:

The program allows integration with tokens from the Token2022 program. However, many parts of the code still use the deprecated `transfer` instruction for transfers. This leads to incompatibility with tokens that have certain extensions, as the transfers may fail.

```
pub fn transfer_spl_tokens(params: TokenTransferParams) -> Result<()> {
    ...
        if let Some(seeds) = signer_seeds {
            #[allow(deprecated)]
            token_interface::transfer(
                CpiContext::new_with_signer(token_program.clone(),
    transfer_ix, seeds),
                amount,
            )?
        } else {
            #[allow(deprecated)]
            token_interface::transfer(CpiContext::new(token_program.clone(),
    transfer_ix), amount)?
        }
    }

    Ok(())
}
```

```
fn deposit_to_liquidity(ctx: &Context<Deposit>, amount: u64, mint: Pubkey)
    -> Result<u64> {
    ...
    // Transfer tokens to liquidity
    transfer_spl_tokens(TokenTransferParams {
```

```
        source: ctx.accounts.depositor_token_account.to_account_info(), //
    depositor token account
        destination: ctx.accounts.vault.to_account_info(), // vault, aka
    liquidity mint PDA
        authority: ctx.accounts.signer.to_account_info(),  // msg.sender
        amount,                                             // amount
        token_program: ctx.accounts.token_program.to_account_info(), //
    token program
        signer_seeds: None,
        mint: None, //
    })?;
    ...
}
```

Specifically, extensions like `TransferHookAccount`, `TransferFeeAmount`, and `PausableAccount` require the mint account to be included in the instruction.

```
pub(crate) fn process_transfer(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    amount: u64,
    transfer_instruction: TransferInstruction,
) → ProgramResult {
    ...
        else {
            // Transfer hook extension exists on the account, but no mint
            // was provided to figure out required accounts, abort
            if source_account
                .get_extension::<TransferHookAccount>()
                .is_ok()
            {
                return Err(TokenError::MintRequiredForTransfer.into());
            }

            // Transfer fee amount extension exists on the account, but no
    mint
            // was provided to calculate the fee, abort
            if source_account
                .get_extension_mut::<TransferFeeAmount>()
                .is_ok()
            {
                return Err(TokenError::MintRequiredForTransfer.into());
            }

            // Pausable extension exists on the account, but no mint
            // was provided to see if it's paused, abort
```

```
            if source_account.get_extension::<PausableAccount>().is_ok() {
                return Err(TokenError::MintRequiredForTransfer.into());
            }

            (0, None, None)
        };
    ...
}
```

Recommendations:

It is recommended to avoid using the deprecated `transfer` instruction. If the mint account is not passed in the instruction, consider retrieving the mint account address from the TokenAccount's data instead.

Fluid: Resolved with @dd1e564cfa4...

Zenith: Verified.

[L-3] The `f_token` metadata will not be initialized.

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

Target

- admin.rs

Description:

Since the `initialize_token_metadata` call is commented out in the `init_lending` instruction, the `f_token` metadata cannot be created.

```
pub fn init_lending(
    ctx: Context<InitLending>,
    symbol: String,
    liquidity_program: Pubkey,
) → Result<()> {
    ...
    // ctx.accounts.initialize_token_metadata(symbol)?;

    Ok(())
}
```

Recommendations:

It is recommended to uncomment the `initialize_token_metadata` call so that the metadata for the `f_token` can be created when the `init_lending` instruction is executed.

Fluid: Resolved with @4cefb7357c4...

Zenith: Verified.

[L-4] Compound interest deviation in borrow exchange price calculation

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

Target

- lending/src/state/token_reserve.rs

Description:

The borrow exchange price calculation in the `calculate_exchange_prices` function uses a simple interest approximation that produces different results based on update frequency, leading to price inconsistencies between different update patterns.

The current implementation accumulates interest linearly per update:

```
borrow_exchange_price = borrow_exchange_price.safe_add(
    borrow_exchange_price
        .safe_mul(borrow_rate)?
        .safe_mul(seconds_since_last_update)?
        .safe_div(SECONDS_PER_YEAR.safe_mul(FOUR_DECIMALS)?)?
)?;
```

This creates divergent outcomes for identical time periods with different update frequencies:

- **1 update after 10 seconds**: Results in price P_1
- **10 updates during 10 seconds**: Results in price P_2 (where $P_2 > P_1$)

The deviation becomes significant with longer update intervals:

- **Per-second updates vs 1 annual update (6% APR)**: 1.061820282945 vs 1.06 (0.17% difference)
- **Per-second updates vs 1 annual update (10% APR)**: 1.105154345224 vs 1.10 (0.47% difference)

While the difference might be insignificant, it becomes more significant with higher borrow rates and is worth considering.

Recommendations:

Consider calculating the new price using a compound interest approximation.

Fluid: Acknowledged

[L-5] No way to close and reclaim rent for some user-related accounts

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

Target

- programs/liquidity/src/state/context.rs
- programs/vaults/src/state/context.rs

Description:

The `UserClaim` accounts must be created by users prior to being passed as accounts for the various operation-related instructions. They cost rent to store, but once the user is done with them, there is no way to close the accounts and reclaim the rent.

User positions have a similar issue in that the `UserPosition` and associated NFT `Mint` accounts are never closed, even after the position is fully closed.

Recommendations:

Provide instructions for closing the accounts.

Fluid: Resolved the `UserClaim` part with @996368afea0.... Acknowledged the user-position-related part of the submission. While we don't plan to support this at initial launch, we'll definitely consider adding it later if and when there's a clear need for it.

Zenith: Verified the `UserClaim` part.

[L-6] Incorrect behavior of `checked_floor_div`

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

Target

- floor_div.rs

Description:

The method `checked_floor_div` is supposed to round numbers down, however it performs this rounding incorrectly. As we can see in the following code:

```
fn checked_floor_div(&self, rhs: $t) → Option<$t> {
    let quotient = self.checked_div(rhs)?;

    let remainder = self.checked_rem(rhs)?;

    if remainder ≠ <$t>::zero() {
        quotient.checked_sub(<$t>::one())
    } else {
        Some(quotient)
    }
}
```

If the remainder is not zero, 1 will be subtracted from the result. This works for negative integers, as for example `-3 / 2` would yield `-1` and `-1` as a remainder, and rounding down means the result should be `-2` (hence subtracted by 1). However, if we divided `3 / 2`, the result is `1` and the remainder is `1`, meaning the result will be subtracted by `1`, resulting in `0`, while `1` should be the correct result. While this could potentially be a disastrous mistake, the function is not used anywhere in the scope for now, hence the low severity rating.

Recommendations:

Consider subtracting one only if the remainder is **less than** zero:

```
            fn checked_floor_div(&self, rhs: $t) -> Option<$t> {
                let quotient = self.checked_div(rhs)?;
```

```
            let remainder = self.checked_rem(rhs)?;

            if remainder ≠ <$t>::zero() {
            if remainder < <$t>::zero() {
                quotient.checked_sub(<$t>::one())
            } else {
                Some(quotient)
            }
        }
```

Fluid: Resolved with @9bea6051f89...

Zenith: Verified

[L-7] Users may be unable to withdraw dust amounts

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

Target

- programs/liquidity/src/module/user.rs

Description:

The solana implementation of the liquidity protocol, unlike the solidity one, rejects any `operate()` instruction where both of the amounts are under `MIN_OPERATE_AMOUNT`. This means that if a user only has a dust amount remaining (e.g. they meant to withdraw all, but accidentally left some dust), they'll be unable to withdraw it without re-depositing enough to cover the minimum. This may be especially problematic for interest-free suppliers, as interest will never push them over the minimum.

Recommendations:

It's not clear what specific risk this threshold is protecting against, but if it is in fact required, withdrawal requests that result in the remaining supply being under this amount should be rejected.

Fluid: Acknowledged. As a general practice across our protocols, we enforce limits on both extremely small and extremely large amounts to mitigate risks such as dust accumulation, unintended donations, or large-value manipulation. In the rare case a user wishes to withdraw the remaining dust, they can simply deposit a small amount and then withdraw their full balance. Also note that by default, for every protocol that we deploy we create a small dust position of a few $ that is the first user and will stay in forever, so no actual user will ever face any issues around being locked as the last user because of some edge case revert or underflow. On EVM we do not have the exact same check, but the checks that revert with `UserModule__OperateAmountInsufficient` target a similar issue there, where the user amount would be too small to cause a change in storage values and is thus rejected. This min operate amount check is intended to replicate a similar effect.

[L-8] Some events may contain incorrect information

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

Target

- programs/liquidity/src/module/admin.rs
- programs/liquidity/src/state/context.rs
- programs/vaults/src/module/admin.rs

Description:

Some fields of emitted events may contain incorrect information:

- `LogUpdateRateDataV1` - `rate_data.token` is never constrained to match `UpdateRateData.mint`
- `LogUpdateRateDataV2` - `rate_data.token` is never constrained to match `UpdateRateData.mint`
- `LogUpdateUserSupplyConfigs` - `user_supply_config.user` is never constrained to match `UpdateUserSupplyConfig.protocol`
- `LogUpdateUserBorrowConfigs` - `user_borrow_config.user` is never constrained to match `UpdateUserBorrowConfig.protocol`
- For the `pre_operate()` ix, there is no constraint forcing the `mint` of each supply to match the `mint` input argument. However, in that case, there is no emit to show the incorrect input argument.
- For the `update_core_settings()` ix, the inputs to `LogUpdateCoreSettings` are multiplied by 10, but all come from the input parameters struct, where they're already all scaled to FOUR_DECIMALS.
- For the `update_oracle()` ix, there is no constraint forcing the `oracle` to match the stored `oracle_program`.

Recommendations:

Add constraints to ensure that the input struct/account fields match the input accounts/arguments

Fluid: Resolved with @ca3cc093234..., @3865cb916aa..., @50187fee83d..., @a78dc328f41..., and @a8166fce948...

Zenith: Verified.

[L-9] Missing a check in `init_lending` to ensure that the `mint` and `f_token_mint` token programs are equal

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

Target

- context.rs

Description:

In the `init_lending` instruction, the `f_token_mint` account is initialized using the provided `token_program`. However, there is no check to ensure that this `token_program` matches the token program associated with the `mint`. If `f_token_mint` and `mint` are created using different token programs, subsequent instructions such as `Deposit` will fail.

```
pub struct InitLending<'info> {
    // @dev Only the auths can initialize the lending
    #[account(mut, constraint = lending_admin.auths.contains(&signer.key())
    @ ErrorCodes::FTokenOnlyAuth)]
    pub signer: Signer<'info>,

    #[account(has_one = liquidity_program.key())]
    pub lending_admin: Account<'info, LendingAdmin>,

    pub mint: InterfaceAccount<'info, Mint>,

    #[account(
        init,
        seeds = [F_TOKEN_MINT_SEED, mint.key().as_ref()],
        bump,
        payer = signer,
        mint::decimals = mint.decimals,
        mint::authority = lending_admin,
        mint::token_program = token_program,
    )]
    pub f_token_mint: Box<InterfaceAccount<'info, Mint>>,
```

Recommendations:

It is recommended to add a constraint to ensure that the `f_token_mint` is initialized using the same token program as the `mint` account.

Fluid: Resolved with @56ccd1d5fab...

Zenith: Verified.

[L-10] User supplies and borrows can't be paused independently

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

Target

- programs/liquidity/src/state/context.rs

Description:

During the `init_new_protocol()` instruction, the `Mints` associated with the supply and borrow positions are not required to be equal. However, during the `pause_user()` instruction, only one `Mint` can be supplied, and it's used for both PDAs, and both supplies are required to be paused at the same time. This behavior differs from the solidity code, which allows either one or both supplies to be paused at a time.

Recommendations:

Introduce a flag to the `pause_user()` instruction indicating that either one or both supplies should be paused, or document the difference, with regard to the solidity code, in a code comment.

Fluid: Resolved with @4e50499e8bd..., @1a255df6bd9..., and @928623a29f0...

Zenith: Verified.

[L-11] Incorrect URI passed to Metaplex

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

Target

- programs/lending/src/state/context.rs

Description:

The hard-coded URI `"https://fluid.io"` is passed to the Metaplex program for all fTokens. The metaplex program expects a per-token URI that points to a JSON document, and neither of these expectations are fulfilled.

Recommendations:

{{ ... }} Construct a per-fToken URI which points to a JSON document with the expected metadata format.

Fluid: Resolved with @4cefb7357c4...

Zenith: Verified.

[L-12] Incorrect `token_reserves_liquidity` account for `init_lending()` ix may require redeployment

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

Target

- programs/lending/src/state/context.rs
- programs/vaults/src/module/admin.rs

Description:

There are no constraints to force the `token_reserves_liquidity` account's owner to match the `liquidity_program` input argument. This means that the lending program used by the `lending_admin` may not match the `token_reserves_liquidity`, which will cause operations involving reserves to be incorrect. Since only one fToken mint can be created for a given underlying mint, and the `init_lending()` ix itself won't fail, in order to create a correct fToken mint the lending program will have to be redeployed.

The vault program has a similar issue in that `initialize_vault_admin()` takes in an arbitrary address as its `liquidity` parameter, but the use of `UserSupplyPosition` and `UserBorrowPosition` in the `Operate` Accounts struct, and the use of `TokenReserve` in the `InitVaultStateContext` Accounts struct means that only the vault-compile-time-program-ID of the liquidity program can be used.

Recommendations:

For the lending program:

```
#[account(has_one = mint)]
#[account(has_one = mint, owner = liquidity_program)]
  pub token_reserves_liquidity: AccountLoader<'info, TokenReserve>,
```

For the vault program, add a check to `initialize_vault_admin()` that ensures that the parameter matches the compile-time-program-ID of the liquidity program.

Fluid: Resolved with @1ac36ee9d29...

Zenith: Verified.

[L-13] The `CLAIM` token `transfer_type` does not support withdrawing and borrowing to different users.

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

Target

- context.rs

Description:

In the `operate` instruction, withdrawing and borrowing to different addresses is allowed, but this is incompatible with the `CLAIM` token `transfer_type` because this method requires authorizing tokens to a single claim account, and only one claim account is defined in the ctx.

```
#[derive(Accounts)]
#[instruction(_supply_amount: i128, _borrow_amount: i128, withdraw_to:
    Pubkey, borrow_to: Pubkey, mint: Pubkey)]
pub struct Operate<'info> {
    ...
    #[account(mut, has_one = mint)]
    pub claim_account: AccountLoader<'info, UserClaim>,

    pub token_program: Interface<'info, TokenInterface>,
    pub associated_token_program: Program<'info, AssociatedToken>,
}
```

Recommendations:

It is recommended to define two `claim` accounts in the ctx to support the case where withdrawing and borrowing addresses are different.

```
pub struct Operate<'info> {
    ...
    #[account(mut, has_one = mint)]
    pub claim_account: AccountLoader<'info, UserClaim>,
```

```
    pub withdraw_claim_account: AccountLoader<'info, UserClaim>,

    #[account(mut, has_one = mint)]
    pub borrow_claim_account: AccountLoader<'info, UserClaim>,

    pub token_program: Interface<'info, TokenInterface>,
    pub associated_token_program: Program<'info, AssociatedToken>,
}
```

Fluid: Resolved with @8e7618716d1... and @3ff7ee74f00...

Zenith: Verified.

[L-14] Missing default Pubkey checks

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

Target

- programs/vaults/src/module/admin.rs
- programs/oracle/src/lib.rs

Description:

The vault program has multiple places where it checks for and rejects the default PubKey, but there are a few places without checks:

- `initialize_vault_admin()` - `liquidity` and `authority`
- `init_vault_config()` - `params.supply_token`, `params.borrow_token`, `params.oracles`, `params.oracle_program`, `params.rebalancer`
- `update_oracle()` - `new_oracle`

The oracle program is missing a check in the `init_admin()` ix.

Recommendations:

Add the missing checks

Fluid: Resolved with @1ac36ee9d29..., @f655ca4d54e..., @ca3cc093234..., and @df2b8741a7c...

Zenith: Verified.

[L-15] Borrow/supply magnification may be applied to exchange price growth prior to vault creation

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

Target

- programs/vaults/src/module/admin.rs

Description:

In the `init_vault_state()` instruction, the `supply/borrow_token_reserves_liquidity` liquidity program `TokenReserve` accounts are loaded, and the values they currently have stored for their `supply/borrow_exchange_price` fields are stored, without modification, to the vault's `VaultState` account. It is possible that the stored value is from a timestamp far in the past, or that the vaules were from very different timestamps in the past. When the actual exchange price for the vault is calculated, it applies magnification factors to the difference in price between the stored price and the current live price. This means that the magnification may be applied to price growth that occurred prior to the vault's creation.

Recommendations:

Use the results of `calculate_exchange_prices()`, rather than using the reserve accounts' fields.

Fluid: Resolved with @f4c77f677ac...

Zenith: Verified.

[L-16] The `withdrawal_cap` is not capped at 100%

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

Target

- programs/vaults/src/module/admin.rs

Description:

The `withdrawal_cap` field is meant to be capped at 100%, but is actually capped at 102.3% due to the copying of a bug in the solidity code. Both the `init_vault_config()` and `update_core_settings()` instructions have the bug, but `update_withdraw_gap()` does not.

Recommendations:

Use `THREE_DECIMALS` rather than `X10` for this parameter's validations.

Fluid: Resolved with @a1acb78118a...

Zenith: Verified.

[L-17] Vaults cannot work with mixed legacy/Token-2022 borrow/supply tokens

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

Target

- programs/vaults/src/module/admin.rs

Description:

The liquidity layer is coded such that both legacy and token-2022 tokens can be used a supply or borrow tokens. The vault code allows these `Mint` accounts to be passed as `InterfaceAccount` accounts, but when it comes to verification of the ATAs, it forces both ATAs to use the same `token_program`, effectively preventing the use of multiple token versions.

Recommendations:

Add `supply_token_program` and `borrow_token_program` accounts to the `Operate` and `Liquidate` Accounts structs.

Fluid: Resolved with @9e5659f380e...

Zenith: Verified.

[L-18] No way to change a vault's authority

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

Target

- programs/vaults/src/module/admin.rs

Description:

In the case of a protocol key compromise, there is no function that allows the changing of each vault's `authority` address.

Recommendations:

Add a function to modify the vault's authority

Fluid: Acknowledged. All program authorities will be set to a team controlled multi sig. Moreover if needed, we can add support for authority migration in a future upgrade.

[L-19] Admin configuration updates can cause limit calculation discrepancies

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

Target

- liquidity/src/module/admin.rs

Description:

The admins have the ability to call the `update_user_supply_config` and `update_user_borrow_config` instructions to update the relevant values in `UserSupplyPosition` or `UserBorrowPosition`, respectively. These values can potentially impact the withdrawal and borrow limits - namely, the values of `expand_pct`, `expand_duration`, and `base_withdrawal_limit` from `UserSupplyPosition` influence the withdrawal limit, while `expand_pct`, `expand_duration`, `base_debt_ceiling`, and `max_debt_ceiling` influence the borrow limit. Since the outcome limit is based on time elapsed between respective liquidity updates, the `last_update` value is tracked in both account types and is updated every time the limits are updated.

However, it is not updated when the key values are changed, which means that the next time the limit should be updated, it will use the newly updated changes, leading to discrepancies and potentially undesired behavior. The following table demonstrates changes to values vs. expected values if `expand_pct` is changed:

action	amount	timestamp	expand_pct	expand_duration	last_updated	base	supply
supply	2000000	1000	5000	1200	0	1000000	2000000
$withdraw_1$	1000000	1200	5000	1200	1000	1000000	1000000
$withdraw_2$	500000	1200	2500	1200	1000	1000000	1500000
$withdraw_3$	12500000	1200	7500	1200	1000	1000000	750000

The table showcases how the limits change when the `expand_pct` field is changed at timestamp 1200 (withdraw₂ and withdraw₃) vs. when it stays the same (withdraw₁).

Recommendations:

Consider updating the withdraw and borrow limits, along with the last update time of these accounts, when calling `update_user_supply_config` or `update_user_borrow_config`.

Fluid: Acknowledged

[L-20] Use of `pause_user()` prior to setting user configuration breaks subsequent configuration

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

Target

- programs/liquidity/src/module/admin.rs

Description:

The `UserSupplyPosition` and `UserBorrowPosition` accounts have their `status` fields set to `2` during initialization, to indicate that configuration has not yet been done. If a guardian or governance ends up calling `pause_user()`, those accounts' `status` will be changed to `1` without storing the fact that it had been `2`. When the user is eventually unpaused, the `status` is changed to `0`, regardless of whether it had been `2` prior to the pause. Having `configs_not_set()` return false due to this change, will mean that initial configuration setting will be skipped.

Recommendations:

Do not allow unconfigured users to be paused.

Fluid: Resolved with @698a5fb3443... and @6223ecf0502...

Zenith: Verified.

[L-21] `get_ticks_from_remaining_accounts()` uses the wrong error code when program ownership checks fail

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

Target

- programs/vaults/src/state/tick.rs

Description:

If the program ownership checks fail for any of the `TickData` PDAs read from `remaining_accounts`, the `get_ticks_from_remaining_accounts()` function errors with `ErrorCodes::VaultBranchOwnerNotValid`, which is only supposed to be used by `get_branches_from_remaining_accounts()`.

Recommendations:

Add a `VaultTickDataOwnerNotValid` error to the `ErrorCodes` enum, and use it in the function.

Fluid: Resolved with @8e0bc0ac949...

Zenith: Verified.

[L-22] `add_debt_to_tick()` never executes the initialization code path

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

Target

- programs/vaults/src/module/admin.rs
- programs/vaults/src/utils/operate.rs

Description:

Unlike the Solidity version of vaults, the `init_tick()` instruction unconditionally initializes `total_ids` to one. This means that when the `operate()` instruction reaches the `add_debt_to_tick()` function, that function always finds that `tick_id` is greater than zero, and thus if the position hasn't been liquidated, the execution flow goes to the if-block rather than the else-block. As the comments indicate, the else block is supposed to be where the first initialization occurs. There are no negative effects of the current processing, however, because `update_tick_has_debt()` ends up being called in both blocks, and this is the only processing that needs to occur for initialization, given the `tick_id` is already non-zero.

Recommendations:

Remove the `tick_id > 0` checks in this function since the result will always be true, and update the comments to reflect the actual flows.

Fluid: Resolved with @e03f7a155f4... and @13b7d96b16d...

Zenith: Verified.

[L-23] The governance signer may lose auth or guardian roles

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

Target

- programs/liquidity/src/module/admin.rs
- programs/lending/src/module/admin.rs

Description:

During initialization, the `init_liquidity()` function adds the `liquidity.authority` to both the `auth_list.auth_users` and `auth_list.auth_guardians`, but there are no checks in `update_auths()` or `update_guardians()` to ensure that the governance address is still present. Unlike the solidity code, it's possible to remove all users from both the auth and guardian roles, permanently removing the ability to pause the program in case of emergencies.

The lending program has an analogous issue with its `update_auths()` instruction.

Recommendations:

Do not allow the governance address to be removed during `update_auths()` and `update_guardians()` for either program.

Fluid: Resolved with @a6fd1b21994... and @6e69529bc96...

Zenith: Verified

[L-24] Unrestricted lending program account in rewards transition

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

Target

- programs/lendingRewardRateModel/src/state/context.rs

Description:

The `transition_to_next_rewards` instruction of the LRRM program accepts any account as the `lending_program`. Although the inline comment says it is checked during the lending program invoke, this is not true, as we can see at the invocation site.

This means that the transition can call into any program, which can mean that the desired update of the specific `Lending` account won't occur, meaning the rewards for the period between lending's last update and the current reward's end time won't be accounted for. The impact is not significant since the `Lending` account is updated frequently with every liquidity operation, hence the low severity rating.

Recommendations:

Consider restricting the `lending_program` to be the correct and expected lending program.

Fluid: Resolved with @709c3d3c8f8...

Zenith: Verified.

[L-25] Prior period rewards may be lost if `LRRM.start_rewards()` is called again

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

Target

- programs/lendingRewardRateModel/src/state/context.rs

Description:

Unlike `TransitionToNextRewards.next()`, which calls `update_rate()` so that the prior period's rewards are checkpointed, the `start()` function does not do any checkpointing of the prior rewards. If there are no new mints or burns for the prior period's duration, when the new `start()` period's settings get configured, the old rewards will never have been doled out. If there are queued rewards, the function does not clear them.

Recommendations:

Call `update_rate()` if there already are settings, or only allow `start()` to be called once (require all subsequent changes to use `next()`). Also consider erroring if there are queued rewards.

Fluid: Resolved with @6e84bcffaed... and @27ffd7ef943...

Zenith: Verified.

[L-26] Open TODOs

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

Target

- programs/oracle/src/modules/pyth.rs
- crates/library/src/math/*.rs

Description:

All open TODOs should be addressed before deployment:

```
// @todo @dev should we add pyth confidence check here?
```

Adding a confidence check is recommended since the aggregation algorithm is essentially a weighted median, which may not actually be the right price to use if all prices are equally spaced, and very far apart from each other. Consider, at least, disallowing new debt positions if the confidence interval is much larger than expected.

```
// @todo Check license and audit for this code
```

Any licensing issues should be resolved as soon as possible.

Recommendations:

Address the TODOs

Fluid: Resolved with @86913c5b14b..., @9e6049b5030..., and @diff-4dd85a1...

Zenith: Verified.

[L-27] Vector field sizes not validated when updating accounts

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

Target

- programs/liquidity/src/state/state.rs
- programs/liquidity/src/module/admin.rs
- programs/lending/src/state/state.rs
- programs/lendingRewardRateModel/src/state/state.rs

Description:

The `AuthorizationList` account uses the `#[max_len]` attribute to allow anchor to calculate the right size to allocate, but the various instructions for updating the account's fields never ensure their sizes fall within the expected bounds. Even though `user_classes` is limited to 100 entries, many more entries can be added, as long as the serialized size of the updated data fits within the account's size. If too many entries are added to `user_classes`, this may permanently prevent further entries from being added to `auth_users` and `guardians`, since once created, user classes cannot be removed.

The `LendingAdmin`'s and `LendingRewardsAdmin`'s `auths` fields are also missing bounds checks during updates, which will result in serialization failure errors, rather than with a descriptive error, when too many entries are added

Recommendations:

Require that the vector lengths respect the allocated maximums, in `update_auths()`, `update_guardians()`, `update_user_class()`, and the `update_auths()` functions for lending and rewards admins.

Fluid: Resolved with @9ad38077b41...

Zenith: Verified.

[L-28] Internal errors are not converted to external ones

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

Target

- programs/library/src/errors.rs
- programs/library/src/token/spl.rs
- programs/vaults/src/invokes/mint.rs

Description:

The various programs in the project rely on the library crate for low-level operations, including math and token transfers. The library crate defines its own `ErrorCodes` enum for the errors it produces, but none of the programs calling these functions convert the library's error codes to their own. This will, for instance, result in `cast()` failures appearing to be `UserClassNotFound` failures, rather than `LibraryCastingFailure` ones, since the IDL only includes the error code enum values from the current program. In addition, the library itself does not convert its own CPI failures to its own error codes, which complicates error handling.

The vault program also does SPL-related operations without converting those CPI failures.

Recommendations:

Use `map_err()` to convert between error enums, or use thiserror's `#[from]` attribute.

Fluid: Acknowledged

4.5 Informational

A total of 12 informational findings were identified.

[I-1] Typos

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

Target

- programs/**/*.rs

Description:

The following typos were identified:

`recepient -> recipient`

```
pub recepient: AccountInfo<'info>,
```

```
pub recepient: Pubkey,
```

```
pub recepient_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
```

```
pub fn claim(ctx: Context<Claim>, mint: Pubkey, recepient: Pubkey) ->
    Result<()> {
```

`liqudity -> liquidity`

```
... // the liqudity PDA owns the authority to transfer the tokens
```

`dvision -> division`

```
// easy to check as that variable is NOT the result of a dvision etc.
```

bororwers -> borrowers

```
// borrowRatio_ ⇒ x of total bororwers paying yield. scale to 1e17.
```

atleast -> at least

```
// ... needs to be atleast 1e73 to overflow max limit of ~1e77 in uint256
```

receipient -> recipient

```
pub receipient_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
```

ratemodel -> rate model

```
/// @dev To read PDA of rewards ratemodel to get_rate instruction
```

assset -> asset

```
/// @dev exchange price for the underlying assset in the liquidity protocol
    (without rewards)
```

resuing -> reusing

```
// This let us save on two extra accounts, hence resuing existing to built
    the correct accounts context.
```

Address if -> Address of

```
pub liquidity_program: Pubkey, // Address if liquidity program
```

to built -> to build

```
// This let us save on two extra accounts, hence resuing existing to built
    the correct accounts context.
```

AccountInfo -> UncheckedAccount

```
// Hence loading them as AccountInfo
```

slightlty -> slightly

```
// partials precision is slightlty above 1e9 so this will make sure that on
    every liquidation atleast 1 partial gets liquidated
```

supplt_position -> supply_position

```
let supplt_position =
    ctx.accounts.vault_supply_position_on_liquidity.load()?;
```

coeffcient -> coefficient

```
pub debt_liquidity: u64, // Debt liquidity at this branch, 56 coeffcient | 8
    exponent
```

pratials -> partials

```
pub fn get_current_pratials_ratio(minima_tick_partials: u32, ratio: u128)
    -> Result<(u128, u128)> {
```

acccount -> account

```
// No init_if_needed, as the acccount supplying must already exist in order
    to supply
```

recepient -> recipient

```
pub recepient: AccountInfo<'info>,
```

receipient -> recipient

```
pub receipient_borrow_token_account: Box<InterfaceAccount<'info,
    TokenAccount>>,
```

sence -> since

```
// sence sending vault_supply_token_account here as a dummy value
```

liquidiation -> liquidation

```
// get liquidiation max limit tick (tick at liquidation max limit ratio)
```

Parms -> Params

```
pub struct InitVaultConfigParms {
```

udpate -> update

```
pub fn udpate_state_at_liq_end(&mut self, tick: i32, branch_id: u32) ->
    Result<()> {
```

substraction -> subtraction

```
// liquidity Exchange Prices always increases in next block. Hence
    substraction with old will never be negative
```

liquidityWithdrawaLimit -> liquidityWithdrawalLimit

```
// (liquidityUserSupply - withdrawalGap - liquidityWithdrawaLimit) should be
    less than user's withdrawal
```

remainining_accounts -> remaining_accounts

```
let remainining_accounts = ctx
```

is_iquidate -> is_liquidate

```
fn get_exchange_rate(&self, nonce: u16, is_iquidate: bool) -> Result<u128> {
```

Recommendations:

We recommend fixing the typos.

Fluid: Resolved with @a0930e2d0af... and @a3392bee9a6...

Zenith: Verified.

[I-2] Dead code

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

Target

- programs/liquidity/src/state/context.rs
- programs/liquidity/src/state/state.rs
- programs/liquidity/src/state/token_reserve.rs
- programs/lending/src/state/state.rs
- programs/lending/src/state/seeds.rs
- programs/lending/src/errors.rs
- programs/vaults/src/errors.rs
- programs/vaults/src/errors.rs
- programs/vaults/src/errors.rs
- programs/vaults/src/events.rs
- programs/vaults/src/state/branch.rs

Description:

The targets listed above point to code that is never executed, which adds unnecessary cognitive load during reviews.

Recommendations:

Remove the unused code, or refactor/rewrite other code to call the functions/use the errors.

Fluid: Resolved with @88d2665d99a... and @6cc13dad136...

Zenith: Verified.

[I-3] Accounts need not be marked as mutable

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

Target

- programs/liquidity/src/state/context.rs
- programs/lending/src/state/context.rs
- programs/vaults/src/state/context.rs
- programs/oracle/src/state/context.rs
- programs/lendingRewardRateModel/src/state/context.rs

Description:

The `UpdateAuth`, `UpdateRevenueCollector`, `UpdateUserClass`, `UpdateUserWithdrawalLimit`, `UpdateUserSupplyConfig`, `UpdateUserBorrowConfig`, `UpdateExchangePrice`,`UpdateRateData`, `UpdateTokenConfig`, `ChangeStatus`, `PauseUser`, `AdminContext`, `UpdateAuths`, `StopLendingRewards`, and `LendingRewards` Accounts structs all include a mutable `authority` account, even though their respective instructions never modify their `authority` accounts. This will lead to unnecessary transaction contention, possibly leading to higher transaction costs required for these instructions. In addition, the `PreOperate.protocol`, `PreOperate.vault`, `Operate.protocol`, `Operate.liqudity`, `Claim.user`, `UpdateRewardsRateModel.signer`, `UpdateAuth.signer`, `UpdateRebalancer.signer`, `Withdraw.rate_model`, `UpdateRate.signer`, `UpdateRate.f_token_mint`, `InitLending.lending_admin`, `InitLending.mint`, `InitPosition.vault_admin`, `Operate.vault_config`, `Operate.tick_id_data`, `Operate.supply_rate_model`, `Operate.borrow_rate_model`, `Operate.liquidity` (once liquidity's `Operate.liquidity` stops being mutable), `Operate.liquidity_program`, `Liquidate.vault_config`, `Liquidate.supply_token`, `Liquidate.borrow_token`, `Liquidate.top_tick_data`, `Liquidate.supply_rate_model`, `Liquidate.borrow_rate_model`, `Liquidate.liquidity` (once liquidity's `Operate.liquidity` stops being mutable), `Liquidate.liquidity_program`, `OracleInit.oracle_admin`, and `LendingRewards.lending_rewards_admin` accounts need not be mutable.

Recommendations:

Remove the `mut` modifier from the listed accounts.

Fluid: Resolved with @dd092438d1d..., @b0881dfdb6b..., and @pull/62

Zenith: Verified.

[I-4] Delayed reward distribution may temporarily prevent withdrawals

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

Target

- helpers.rs

Description:

For a given token's lending, in addition to earning interest from the liquidity, extra rewards can also be flexibly configured through the `lendingRewardRateModel`. These rewards increase the price of the `f_token` over time.

```
fn calculate_new_token_exchange_price(
    new_liquidity_exchange_price: u64,
    lending: &Account<Lending>,
    current_rate_model: &Account<LendingRewardsRateModel>,
    f_token_total_supply: u64,
) → Result<(u64, bool)> {
    ...
    let total_return_in_percent = total_return_from_rewards.safe_add(
        new_liquidity_exchange_price
            .safe_sub(old_liquidity_exchange_price)?
            .cast::<u128>()?
            .safe_mul(RETURN_PERCENT_PRECISION)?
            .safe_div(old_liquidity_exchange_price.cast()?)?
            .cast()?,
    )?;

    let new_token_exchange_price: u64 = old_token_exchange_price.safe_add(
        old_token_exchange_price
            .cast::<u128>()?
            .safe_mul(total_return_in_percent.cast()?)?
            .safe_div(RETURN_PERCENT_PRECISION)?
            .cast()?,
    )?;

    Ok((new_token_exchange_price, rewards_ended))
```

```
}
```

However, the reward tokens are not distributed when the rewards are enabled. Instead, they are distributed through an off-chain bot via the `rebalance` instruction. If the distribution is delayed, it could, in extreme cases, result in the position not holding enough tokens to allow all users to withdraw.

Recommendations:

It is recommended to improve the reward distribution method.

Fluid: Acknowledged

[I-5] The `Operate` instruction may be called with some accounts that are not used.

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

Target

- context.rs

Description:

When only supplying tokens, the `withdraw_to_account`, `borrow_to_account`, and `claim_account` are not used. If the `transferType` is not `CLAIM`, there is no need to pass in the `claim_account`. However, every time `Operate` is called, these accounts are still required to be passed in, which may cause inconvenience.

```
pub struct Operate<'info> {
    ...
    #[account(
        mut,
        associated_token::mint = mint,
        associated_token::authority = withdraw_to,
        associated_token::token_program = token_program
    )]
    pub withdraw_to_account: Box<InterfaceAccount<'info, TokenAccount>>,

    #[account(
        mut,
        associated_token::mint = mint,
        associated_token::authority = borrow_to,
        associated_token::token_program = token_program
    )]
    pub borrow_to_account: Box<InterfaceAccount<'info, TokenAccount>>,

    #[account(mut, has_one = mint)]
    pub claim_account: AccountLoader<'info, UserClaim>,

    pub token_program: Interface<'info, TokenInterface>,
    pub associated_token_program: Program<'info, AssociatedToken>,
}
```

Recommendations:

It is recommended to make these accounts optional by using `Option`, allowing them to be provided only when needed.

Fluid: Resolved with @bdf526de478...

Zenith: Verified.

[I-6] Unused instruction accounts

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

Target

- programs/liquidity/src/state/context.rs
- programs/lending/src/state/context.rs
- programs/vaults/src/state/context.rs

Description:

The `update_exchange_price()` requires an `authority` signer, but the instruction is meant to be permissionless, and thus the account is never checked against anything.

The `InitLending.associated_token_program`, `UpdateRate.signer`, `Deposit.borrow_token_reserves_liquidity`, and `Liquidate.vault_admin` accounts are also unused.

The `Liquidate.top_tick_data` doesn't have any constraints that force it to match the topmost tick, so any `TickData` from the vault can be passed. Outside of being loaded to confirm ownership and PDA validity in `verify_liquidate()`, it's not used for anything.

Recommendations:

Remove the unused accounts.

Fluid: Resolved with @6e523fbe7e1..., @acb79b768db..., and @3bcc21a8edf...

Zenith: Verified.

[I-7] Unused account fields waste rent

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

Target

- programs/liquidity/src/state/state.rs
- programs/liquidity/src/state/rate_model.rs
- programs/liquidity/src/state/token_reserve.rs
- programs/liquidity/src/state/user_borrow_position.rs
- programs/liquidity/src/state/user_supply_position.rs
- programs/lending/src/state/state.rs
- programs/vaults/src/state/vault_state.rs
- programs/vaults/src/state/branch.rs
- programs/vaults/src/state/tick_has_debt.rs
- programs/vaults/src/state/vault_state.rs
- programs/vaults/src/state/tick.rs
- programs/vaults/src/state/position.rs
- programs/oracle/src/state/state.rs
- programs/lendingRewardRateModel/src/state/state.rs

Description:

The `RateModel`, `AuthorizationList`, `UserClaim`, `TokenReserve`, `UserBorrowPosition`, `UserSupplyPosition`, `BranchData`, `TickHasDebtArray`, `VaultState`, `TickData`, `UserPosition`, `Oracle`, `LendingRewardsAdmin`, and `LendingRewardsRateModel` accounts all contain a `bump` field which costs rent and adds CU overhead, but isn't actually used anywhere. `Lending.borrow_position_on_liquidity` and `VaultState.total_positions` are also unused.

The `LendingRewardsAdmin.lending_program` is also unused. It looks as though the `update_rate()` function is meant to work with any lending program rather than only the compile-time one. If so, this field should be removed. If not, then the various instructions calling that function should constrain the input arg to match that member, and the CPI should be done with the crate's CPI method rather than the generic `invoke()`.

Recommendations:

Remove the fields from the listed structs, and the code that sets their values.

Fluid: Resolved with @5921357d276... and @0946eab44bc.... Acknowledging `LendingRewardsAdmin.bump`, `Oracle.bump`, and `LendingRewardsRateModel.bump`.

Zenith: Verified resolved instances.

[I-8] The `init_token_reserve` instruction does not check whether the token decimals are supported

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

Target

- admin.rs

Description:

The `init_token_reserve` instruction initializes the `token_reserve` and `rate_model` accounts for the corresponding token but does not call `check_token_decimals_range` to verify whether the token decimals are supported. This check is only performed later when the `update_rate_data` instruction is called.

```
pub fn init_token_reserve(context: Context<InitTokenReserve>) -> Result<()>
    {
    let mut token_reserve = context.accounts.token_reserve.load_init()?;

    token_reserve.mint = context.accounts.mint.key();
    token_reserve.vault = context.accounts.vault.key();
    token_reserve.bump = context.bumps.token_reserve;
    token_reserve.last_update_timestamp
    = Clock::get()?.unix_timestamp.cast()?;
    token_reserve.supply_exchange_price = EXCHANGE_PRICES_PRECISION.cast()?;
    token_reserve.borrow_exchange_price = EXCHANGE_PRICES_PRECISION.cast()?;

    let mut rate_model = context.accounts.rate_model.load_init()?;

    rate_model.mint = context.accounts.mint.key();
    rate_model.bump = context.bumps.rate_model;

    Ok(())
}

pub fn update_rate_data_v1(
    context: Context<UpdateRateData>,
    rate_data: RateDataV1Params,
) -> Result<()> {
```

```
    check_token_decimals_range(&context.accounts.mint)?;
    ...
}
```

If `token_reserve` and `rate_model` accounts are initialized for an unsupported token, the issue will only be detected upon calling `update_rate_data`, resulting in wasted rent and transaction fees for the admin.

Recommendations:

It is recommended to call `check_token_decimals_range` during the `init_token_reserve` instruction to verify whether the token decimals are supported.

Fluid: Resolved with @62dd8a70cab...

Zenith: Verified.

[I-9] Misleading comments

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

Target

- programs/lending/src/state/context.rs
- programs/vaults/src/state/vault_config.rs

Description:

The following comments are misleading, which adds unnecessary cognitive load during reviews:

- In the `Deposit` and `Withdraw` structs, the address constraint is on the `lending_admin` member, but for this instruction they were moved, but the comment was never removed:

```
#[account(mut, address = lending_admin.liquidity_program)]
/// CHECK: Safe, we check the address in the lending_admin PDA
pub liquidity_program: AccountInfo<'info>,
```

- The instruction's `supply_token_reserves_liquidity` is never used in any CPI calls:

```
#[account(has_one = mint)]
/// CHECK: Safe as this will be verified in liquidity program CPI call
pub supply_token_reserves_liquidity: AccountLoader<'info, TokenReserve>,
```

- This comment looks like it was a left-over from a refactoring:

```
pub oracle: Pubkey,             // Oracle PDA hops, maximum 4
```

Recommendations:

Update the comments to reflect the current behavior.

Fluid: Resolved with @6e0cb693443... and @35abf6da668...

Zenith: Verified.

[I-10] Unhandled `SKIP` token `transfer_type`

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

Target

- token.rs

Description:

The `transfer_type` enum defines three types.

```
pub enum TransferType {
    SKIP,   // skip transfer
    DIRECT, // transfer directly to the user (no claim)
    CLAIM,  // transfer to claim account and then can be claimed by user
    later
}
```

However, in `handle_transfer_or_claim` only the other two types of transfers are handled. The `SKIP` type transfer is not processed, which may cause errors when this type of transfer is used in integration.

```
pub fn handle_transfer_or_claim(
    transfer_type: &TransferType,
    claimer: Pubkey,
    claim_account: &mut AccountLoader<UserClaim>,
    last_stored_claim_amount: u64,
    transfer_params: TokenTransferParams,
) -> Result<u64> {
    ...
    match transfer_type {
        TransferType::DIRECT => {
            transfer_spl_tokens(transfer_params)?;
            Ok(0)
        }
        TransferType::CLAIM => {
            let mut claim_account = claim_account.load_mut()?;
```

```
            if claim_account.user ≠ claimer {
                return Err(ErrorCodes::InvalidUserClaim.into());
            }

            claim_account.approve(transfer_params.amount)?;
            return Ok(transfer_params.amount);
        }
        _ ⇒ return Err(ErrorCodes::InvalidTransferType.into()),
    }
}
```

Recommendations:

It is recommended to either remove the `SKIP transfer_type` or implement the corresponding handling logic.

Fluid: Acknowledged

[I-11] There may not be enough user class entries to support future tokens

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

Target

- programs/liquidity/src/state/state.rs

Description:

The `AuthorizationList.user_classes` account field is meant to track which liquidity program users are allowed to be paused. If there is no entry, it defaults to allowed, but if it's set to one for a user, that user cannot be paused. For every fToken (one per liquidity supply token) and for every vault (one per supply-and-borrow token pair), a new protocol/user is created in the liquidity program. Currently, according to fluid.io, for Solidity there are six fTokens, and 85 vaults. If the Solana version of fluid is as successful or greater over time, and a lot of user classes need to be set to unpauseable, the `AuthorizationList` PDA may run out of space.

Recommendations:

Significantly increase the size of the `user_classes` field.

Fluid: Acknowledged. This feature is rarely used we typically don't list users as un-pausable, and by default all users can be paused, which is the intended behavior. Given this, the current field size has been sufficient. If ever required, we can expand the field in a future upgrade.

[I-12] Rewards calculation does not credit interest-related TVL growth

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

Target

- programs/lending/src/utils/helpers.rs

Description:

The rewards calculation for both the prior (or current) and the current (or next period), use the TVL calculated using the prior exchange prices. Consider the case where the TVL is right above the cutoff, but then a user withdraws such that the TVL is now just below the limit. If there are no mints or burns until after the queued rewards period has elapsed, there will be zero rewards even though it is likely that, due to interest growth, TVL will grow above the limit without a new mint needing to occur.

Recommendations:

If this is expected behavior, it might be worth while to document in a code comment that the `update_rate()` instruction can be used to get credit for TVL growth due to interest.

Fluid: We've added the comment here: @e143c8abe75.... The `start_tvl` is only used to set some minimal realistic amount to avoid any weird edge cases, by default this has always been set to ~$1k worth of the token. Realistically, we can assume that any fToken that gets rewards, has a TVL way above the configured `start_tvl`. Using the older exchange price is then actually beneficial to users as they get accounted a slightly higher rewards rate. Also, we can assume frequent interactions on any fToken, especially on one that has active rewards going on.

Zenith: Verified.