Pashov Audit Group

Coalesce Finance Security Review

October 13th 2025 - October 24th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Coalesce Finance

Coalesce Finance is a fork of Solend - protocol for lending and borrowing on Solana.

5. Executive Summary

A time-boxed security review of the **saguarocrypto/coalescefi-program** repository was done by Pashov Audit Group, during which **FrankCastle**, **newspace**, **0xdeadbeef**, **ZeroTrust01** engaged to review **Coalesce Finance**. A total of **32** issues were uncovered.

Protocol Summary	
Project Name	Coalesce Finance
Protocol Type	Lending
Timeline	October 13th 2025 - October 24th 2025

Review commit hash:
- ed134a00f3a72bce725e56de283b15940bdd94dd (saguarocrypto/coalescefi-program)

Fixes review commit hash:
- 56d03be899d26c756faf30ba9e580395481e5cf9 (saguarocrypto/coalescefi-program)

Scope

`coalescefi_constants.rs` `coalescefi_helpers.rs` `coalescefi_instruction.rs` `coalescefi_macros.rs` `coalescefi_pda_utils.rs` `coalescefi_processor.rs` `entrypoint.rs` `lib.rs` `multi_pool.rs` `processor.rs` `constants.rs` `legacy_compat.rs` `platform_state.rs` `mod.rs` `pool_ownership.rs` `pool_rate_config.rs` `user_state.rs`

6. Findings

Findings count

Severity	Amount
High	3
Medium	14
Low	15
Total findings	**32**

Summary of findings

ID	Title	Severity	Status
[H-01]	Blacklisted users can withdraw via combined handler	High	Resolved
[H-02]	`penalty_amount` incorrectly used as principal repayment	High	Resolved
[H-03]	`u16` to `u8` cast inflates repayment penalty fees	High	Resolved
[M-01]	Empty deposit reserve accounts array in whitelist-only pools causes runtime error	Medium	Resolved
[M-02]	Blacklist bypass in `process_withdraw_obligation_collateral()`	Medium	Resolved
[M-03]	Inconsistent time precision in penalty calculations	Medium	Resolved
[M-04]	Single-borrower enforcement blocks whitelisted non-owners from borrowing	Medium	Resolved
[M-05]	Fixed-term expiry bypass	Medium	Resolved
[M-06]	Platform pause guard missing across several privileged instructions	Medium	Resolved
[M-07]	`days_late` handling inconsistency in grace period	Medium	Resolved
[M-08]	`BorrowObligationLiquidity` lacks CoalesceFi extension accounts	Medium	Resolved

ID	Title	Severity	Status
[M-09]	`InitObligation` account order error disrupts instruction	Medium	Resolved
[M-10]	`InitMultiPoolReserve` accounts fall short of processor needs	Medium	Resolved
[M-11]	`BorrowObligationLiquidity` account needs mismatch processor logic	Medium	Resolved
[M-12]	`CoalesceFi` instruction discriminants misroute commands	Medium	Resolved
[M-13]	Platform pause checks can be skipped, leaving sensitive ops enabled	Medium	Resolved
[M-14]	`accrue_interest` neglects fixed rate settings	Medium	Resolved
[L-01]	u64::MAX borrowing disabled by input validation	Low	Resolved
[L-02]	Redundant `can_borrow()` validation in borrow operation	Low	Resolved
[L-03]	Unreachable auto-whitelist logic in `process_init_reserve()`	Low	Resolved
[L-04]	Hardcoded staleness threshold in pyth price	Low	Resolved
[L-05]	Incorrect platform pause check	Low	Resolved
[L-06]	Inconsistent token limit constant and lack of validation in `set_access_mode()`	Low	Resolved
[L-07]	Duplicate borrow limit validation in `process_borrow_obligation_liquidity()`	Low	Resolved
[L-08]	Inefficient token limit validation in `process_borrow_obligation_liquidity()`	Low	Resolved
[L-09]	`PoolOwnership` PDA owner not checked	Low	Resolved
[L-10]	`platform_config.pool_reserve` remains none	Low	Resolved
[L-11]	Init reserve allows AC incomplete pools	Low	Resolved
[L-12]	Platform fee receiver update does not affect existing reserves	Low	Resolved
[L-13]	Missing per-reserve access control `PDA` enforces global whitelist	Low	Resolved
[L-14]	Borrow flow requires whitelist for every user	Low	Resolved

ID	Title	Severity	Status
[L-15]	`ACCESS_MODE_NO_ACCESS` incorrectly grants borrow privileges	Low	Resolved

High findings

[H-01] Blacklisted users can withdraw via combined handler

Severity

Impact: High

Likelihood: Medium

Description

The combined handler `process_withdraw_obligation_collateral_and_redeem_reserve_liquidity` (`program/src/processor.rs:3645-3698`) skips the `UserState::is_blacklisted` guard that the standalone withdrawal path runs before calling `_withdraw_obligation_collateral`.

Reproduction outline: 1. Create a `UserState` for a borrower and set `is_blacklisted = true`.
2. Ensure the borrower has collateral deposited in an obligation, and the reserve still has liquidity to redeem.
3. Invoke the `WithdrawObligationCollateralAndRedeemReserveLiquidity` instruction, supplying the usual accounts but omitting (or spoofing) the blacklist PDA.
4. The transaction succeeds:
- `_withdraw_obligation_collateral` never sees a blacklist error because the caller performed no pre-check.
- `_redeem_reserve_collateral` immediately returns liquidity to the user.

This allows globally blocked users to unwind positions and extract funds despite compliance requirements mandating the block.

Recommendations

- Mirror the standalone handler: look up `get_user_state_address(program_id, obligation_owner)` and reject when `is_blacklisted` is true.
- Fail fast if the PDA is missing, owned by another program, or cannot be decoded.

[H-02] `penalty_amount` incorrectly used as principal repayment

Severity

Impact: High

Likelihood: Medium

Description

```
  let CalculateRepayResult {
        settle_amount,
        repay_amount,
    } = repay_reserve.calculate_repay(
@>        liquidity_amount.saturating_add(penalty_amount),
        liquidity.borrowed_amount_wads,
    )?;
```

```
 pub fn calculate_repay(
        &self,
        amount_to_repay: u64,
        borrowed_amount: Decimal,
    ) -> Result<CalculateRepayResult, ProgramError> {
        let settle_amount = if amount_to_repay == u64::MAX {
@>            borrowed_amount
        } else {
@>            Decimal::from(amount_to_repay).min(borrowed_amount)
        };
        let repay_amount = settle_amount.try_ceil_u64()?;

        Ok(CalculateRepayResult {
            settle_amount,
            repay_amount,
        })
    }
```

Inside `process_repay_obligation_liquidity` (`program/src/processor.rs:3256-3262`), `penalty_amount` is added to `liquidity_amount` before calling `calculate_repay` . The resulting `amount_to_repay` feeds both `settle_amount` and `repay_amount` , so `obligation.repay()` subtracts the penalty from the principal. If the penalty exceeds the outstanding debt, it is clipped by `min(amount_to_repay, borrowed_amount)` and effectively never collected. The SPL token transfer later sends only `repay_amount` , with no separate movement for the penalty, confirming it is not charged independently.

Recommendations

Split the flows: compute principal repayment solely from `liquidity_amount` , transfer `penalty_amount` separately to the designated recipient, and ensure it does not reduce the obligation's principal.

[H-03] `u16` to `u8` cast inflates repayment penalty fees

Severity

Impact: High

Likelihood: Medium

Description

`calculate_repayment_penalty` casts the `u16` basis-point values returned by `PoolRateConfig::get_early_penalty()` / `get_late_penalty()` down to `u8` before calling `Decimal::from_percent` (`program/src/processor.rs:345` and `:354`).

```
let penalty_rate = Decimal::from_percent(pool_config.get_early_penalty() as u8);
```

Any penalty above 255 bps wraps during the `u16` → `u8` cast—e.g., 1000 bps becomes 232 bps and is applied as 2.32%, while 500 bps becomes 2.44%. Borrowers are drastically overcharged, and the computation risks overflow, breaking repayment flows.

Recommendations

Preserve the full `u16` precision when building the penalty rate: add a safe helper that converts basis points to `Decimal` (divide by 10,000 or use an explicit `from_bps` API) and call it from `calculate_repayment_penalty` instead of casting to `u8` . Add regression tests with penalties above 255 bps to guarantee no future precision loss.

Medium findings

[M-01] Empty deposit reserve accounts array in whitelist-only pools causes runtime error

Severity

Impact: High

Likelihood: Low

Description

The `process_borrow_obligation_liquidity()` incorrectly passes an empty array to `update_borrow_attribution_values()` when `loan_to_value_ratio = 0` (whitelist-only pools), potentially causing runtime errors and incorrect attribution calculations. When `loan_to_value_ratio = 0` (whitelist-only pools), the code incorrectly assumes no deposit reserve accounts are needed, even when the obligation has existing deposits that require attribution updates. This causes runtime errors in whitelist-only pools with existing deposits.

```
// Lines 2998-3010
let deposit_reserve_accounts = if reserve_loan_to_value_ratio > 0 && accounts.len() > 14 {
    // Overcollateralized pool - deposit reserves expected
    &accounts[14..]
} else {
    // Whitelist-only pool (0% LTV) - no deposit reserves needed
    &[]  // ← PROBLEM: Empty array even when deposits exist
};

// Lines 3011-3015
let (open_exceeded, _) = update_borrow_attribution_values(
    &mut obligation,
    deposit_reserve_accounts,  // ← Empty array passed here
    Some(borrow_reserve_info),
)?;
```

Recommendations

Skip attribution update for whitelist-only pools.

[M-02] Blacklist bypass in `process_withdraw_obligation_collateral()`

Severity

Impact: Medium

Likelihood: Medium

Description

In `process_withdraw_obligation_collateral()`, the blacklist check is conditional. If the `global_access_pda` is NOT provided in `remaining_accounts`, the blacklist check is completely bypassed. Even in `process_withdraw_obligation_collateral_and_redeem_reserve_liquidity()`, there is no blacklist check at all. Thus, blacklisted users can withdraw collateral from their obligations.

Recommendations

Require the global access account to always be provided, or fetch the account data directly if not provided in `remaining_accounts`

[M-03] Inconsistent time precision in penalty calculations

Severity

Impact: Medium

Likelihood: Medium

Description

Early Penalty: Used seconds for precise calculation. Late Penalty: Used days (truncated), causing precision loss.

```
// Early case - PRECISE (seconds)
let time_remaining = maturity - current_time;  // seconds
let time_factor = Decimal::from(time_remaining as u64)
    .try_div(Decimal::from((pool_config.term_days as u64) * 86400))?;

// Late case - IMPRECISE (days)
let days_late = ((current_time - maturity) / 86400) as u64;  // truncates!
```

Examples of the problem: - 86399 seconds late: days_late = 86399 / 86400 = 0 → NO PENALTY (wrong!). - 1 second over day boundary: days_late = 1 → FULL DAY PENALTY (wrong!).

Recommendations

Use seconds_late instead of days_late, same as an early penalty.

[M-04] Single-borrower enforcement blocks whitelisted non-owners from borrowing

Severity

Impact: Medium

Likelihood: Medium

Description

Docs and code suggest pool creators have guaranteed access, while whitelisted non-owners can also borrow if they are whitelisted in their respective `UserState`.

The program persists a `borrower_pubkey` on reserve init and enforces that only that user can borrow (unless default/pubkey-zero).

```
if borrow_reserve.borrower_pubkey != Pubkey::default()
    && borrow_reserve.borrower_pubkey != *obligation_owner_info.key
{
    return Err(LendingError::InvalidAccountInput.into());
}
```

Where `obligation_owner_info` is the signer.

Later in the code there is a section that checks if the user is whitelisted however, that code is never reached because the above snippet will return an error.

Recommendations

If the signer is not the creator - check if signer is whitelisted before returning an error.

[M-05] Fixed-term expiry bypass

Severity

Impact: Medium

Likelihood: Medium

Description

The combined handler `process_deposit_reserve_liquidity_and_obligation_collateral` (`program/src/processor.rs:2137-2201`) mints collateral by calling `_deposit_reserve_liquidity`, yet it never loads the pool config nor invokes `check_pool_expiry(..., PoolOperation::Deposit, …)`—the checks that `process_deposit_reserve_liquidity` performs before minting.

Reproduction outline: - Configure a reserve with `PoolRateConfig.term_days > 0`, turning it into a fixed-term pool.
- Let `clock.unix_timestamp` exceed `creation_timestamp + term_days * 86400`, so the pool is expired.
- Invoke the combined instruction `DepositReserveLiquidityAndObligationCollateral` for that reserve. The transaction still succeeds:
- `_deposit_reserve_liquidity` proceeds without the expiry guard and returns newly minted collateral.
- `_deposit_obligation_collateral` immediately deposits it into the obligation.

This allows users to continue topping up obligations after maturity, preventing forced unwinds and undermining the risk model that relies on deposits being blocked once a term pool expires.

Recommendations

- Mirror the standalone deposit flow inside this handler: parse the remaining accounts, load the `PoolRateConfig`, and run `check_pool_expiry(..., PoolOperation::Deposit, …)` before minting.

[M-06] Platform pause guard missing across several privileged instructions

Severity

Impact: Medium

Likelihood: Medium

Description

The emergency pause is supposed to stop critical fund movements, yet several handlers never invoke `check_platform_pause`, so `PlatformState::is_paused == true` does not block them:

- `process_withdraw_obligation_collateral_and_redeem_reserve_liquidity` (`program/src/processor.rs:3648`)
- `process_deposit_obligation_collateral` (`program/src/processor.rs:2028-2064`) lets users keep adding collateral.
- `process_redeem_fees` (`program/src/processor.rs:3861-3924`) allows fee redemption while paused.
- `process_flash_borrow_reserve_liquidity` (`program/src/processor.rs:3928-3955`) and `process_flash_repay_reserve_liquidity` (`program/src/processor.rs:4086-4117`) permit flash loans regardless of pause state.
- `process_forgive_debt` (`program/src/processor.rs:4242-4316`) continues forgiving debt in a paused market.

Other sensitive paths (deposit/withdraw reserve, borrow, liquidation) already call `check_platform_pause`; the inconsistency leaves a sizable attack surface where paused markets can still mutate state or move funds.

Recommendations

- Add `check_platform_pause(program_id, accounts)` at the top of each affected instruction; fail closed if the PDA is missing or paused.

[M-07] `days_late` handling inconsistency in grace period

Severity

Impact: Medium

Likelihood: Medium

Description

```
fn calculate_repayment_penalty(
    pool_config: &PoolRateConfig,
    repay_amount: u64,
    borrow_start: i64,
    current_time: i64,
) -> Result<u64, ProgramError> {
    if pool_config.term_days == 0 {
        return Ok(0); // No penalties for perpetual pools
    }
```

```
    let maturity = borrow_start + (pool_config.term_days as i64 * 86400);

    if current_time < maturity {
        // Early repayment penalty
        let time_remaining = maturity - current_time;
        let time_factor = Decimal::from(time_remaining as u64)
            .try_div(Decimal::from((pool_config.term_days as u64) * 86400))?;
        let penalty_rate = Decimal::from_percent(pool_config.get_early_penalty() as u8);

        let penalty = Decimal::from(repay_amount)
            .try_mul(penalty_rate)?
            .try_mul(time_factor)?
            .try_floor_u64()?;
        Ok(penalty)
    } else if current_time > maturity + (pool_config.grace_period_days as i64 * 86400) {
        // Late payment penalty
@>        let days_late = ((current_time - maturity) / 86400) as u64;
        let penalty_rate = Decimal::from_percent(pool_config.get_late_penalty() as u8);

        let penalty = Decimal::from(repay_amount)
            .try_mul(penalty_rate)?
            .try_mul(Decimal::from(days_late))?
            .try_div(Decimal::from(365u64))?
            .try_floor_u64()?;
        Ok(penalty)
    } else {
        Ok(0) // Within grace period
    }
}
```

```
 if let Some(config) = pool_config.as_ref() {
        // Use pool config for penalty calculation if available
        // Use actual origination timestamp from obligation metadata
        let borrow_start = if let Some(meta) = liquidity.fixed_term_metadata() {
            if meta.is_active() {
                meta.origination_timestamp
            } else {
                // No fixed-term metadata, skip penalty calculation
                clock.unix_timestamp
            }
        } else {
            // No fixed-term metadata, skip penalty calculation
            clock.unix_timestamp
        };

@>        penalty_amount = calculate_repayment_penalty(
            config,
            liquidity_amount,
            borrow_start,
            clock.unix_timestamp,
        )?;
    } else {
        // Fall back to per-borrow metadata if no pool config
        let fixed_term_snapshot = liquidity.fixed_term_metadata().cloned();

        if let Some(meta) = fixed_term_snapshot {
```

```
            // ⚠️ COALESCEFI: Enhanced penalty calculation with overflow protection
            let maturity = meta.maturity_timestamp();

            // Early repayment penalty - fail on overflow instead of returning 0
            if clock.unix_timestamp < maturity {
                penalty_amount = (liquidity_amount as u128)
                    .checked_mul(meta.early_penalty_bps as u128)
                    .and_then(|v| v.checked_div(10_000))
                    .and_then(|v| u64::try_from(v).ok())
                    .ok_or(LendingError::MathOverflow)?;
            } else if clock.unix_timestamp > meta.grace_period_end() {
                // Late penalty with bounded time calculation
                const MAX_PENALTY_DAYS: i64 = 365; // Cap at 1 year
    @>            let days_late = ((clock.unix_timestamp - meta.grace_period_end()) / 86400)
                    .min(MAX_PENALTY_DAYS);

                // Calculate late penalty with time factor (days late / max days)
                let time_factor = (days_late as u128) * 10_000 / (MAX_PENALTY_DAYS as u128);

    @>            penalty_amount = (liquidity_amount as u128)
                    .checked_mul(meta.late_penalty_bps as u128)
                    .and_then(|v| v.checked_mul(time_factor))
                    .and_then(|v| v.checked_div(10_000 *
10_000)) // Divide by both basis points and time factor scaling
                    .and_then(|v| u64::try_from(v).ok())
                    .ok_or(LendingError::MathOverflow)?;
            }
        }
    }
```

Compute “days late” differently:

PoolRateConfig branch (`calculate_repayment_penalty`)

- First checks `current_time > maturity + grace_period` before treating the loan as late.
- Computes `days_late = (current_time - maturity) / 86400`. Because that check already guarantees `current_time` exceeds `maturity + grace_period`, the grace period is implicitly counted. The formula effectively becomes “actual late days + grace-period days.”
- A separate proportional adjustment is applied for early repayment based on the remaining term.

Fallback branch (using the fixed-term metadata)

- Compares `clock.unix_timestamp` with `meta.grace_period_end()`; only timestamps beyond the grace period qualify as late.
- Uses `days_late = (current - grace_period_end) / 86400`, so late-days start after the grace window.
- Early-repayment penalties are keyed off maturity without additional weighting.

Therefore, the PoolRateConfig branch unintentionally charges for the grace-period days, while the fallback branch does not. This discrepancy can inflate the late fee whenever a pool is configured with `PoolRateConfig`.

Recommendations

Align both paths by calculating `days_late = (current_time - maturity ) / 86400`.

[M-08] `BorrowObligationLiquidity` lacks CoalesceFi extension accounts

Severity

Impact: Medium

Likelihood: Medium

Description

The builder at sdk/src/instruction.rs:1840 creates a BorrowObligationLiquidity instruction that includes only nine standard accounts (liquidity source/destination, borrow reserve, liquidity fee receiver, obligation, lending market, lending market authority, obligation owner, and the SPL Token program), plus the collateral_reserves list and an optional host_fee_receiver. Every one of these accounts is provided directly by the caller.

Meanwhile, process_borrow_obligation_liquidity in program/src/processor.rs:2448 expects, after reading those nine accounts, that any remaining accounts form a "CoalesceFi extension block." It loads the platform config, pool ownership, access control, user whitelist, token limit, and other PDAs at fixed positions—for example, remaining_accounts_slice.get(0) must be the platform config PDA, and lines 2693–2737 call helpers such as get_pool_ownership_at_position and get_access_control_at_position. Later, at lines 3110–3124, it writes platform fees back into PlatformState, so the platform config account must be writable.

The SDK instruction never supplies these CoalesceFi extension accounts. The entries in collateral_reserves are reserve accounts whose layout is incompatible with PlatformState or PoolOwnership, and attempting to unpack them throws errors. Worse, if the optional host_fee_receiver is present, it shifts the positions of all following accounts even further out of alignment.

As a result, process_borrow_obligation_liquidity fails immediately when it tries to read remaining_accounts_slice.get(0), returning ProgramError::NotEnoughAccountKeys, or later when it attempts to deserialize a PlatformState / PoolOwnership and encounters LendingError::InvalidAccountInput or InvalidAccountData. Consequently, the processor cannot handle instructions produced directly by the current SDK.

Recommendations

Explicitly add the required CoalesceFi PDAs (platform config, pool ownership, access control, whitelist, token limit, etc.) to the SDK instruction builder, and supply them in the exact order and with the read/write flags that the on-chain processor expects.

[M-09] `InitObligation` account order error disrupts instruction

Severity

Impact: Medium

Likelihood: Medium

Description

The SDK builder (`sdk/src/instruction.rs:1628`) pushes only five accounts for `init_obligation` : `obligation` , `lending_market` , `obligation_owner` , `rent` , and `token_program` . The on-chain `process_init_obligation` (`program/src/processor.rs:1708`) still expects six accounts—the 4th should be `sysvar::clock` , the 5th `sysvar::rent` , and the 6th the SPL Token Program. Because the SDK omits `clock` , the processor misreads the token program as rent and then fails with `ProgramError::NotEnoughAccountKeys` when it seeks the sixth account, making the instruction unusable.

Recommendations

Bring the client layout back in sync: insert `sysvar::clock::id()` before rent and append the SPL Token Program as the final account. Add integration coverage that constructs the instruction via the SDK and runs it against `process_init_obligation` to prevent future ordering regressions.

[M-10] `InitMultiPoolReserve` accounts fall short of processor needs

Severity

Impact: Medium

Likelihood: Medium

Description

The SDK builder at `sdk/src/instruction.rs:1237` supplies only 13 accounts for `InitMultiPoolReserve` , starting with the pool owner signer and reserve PDAs.

```
0  pool_owner (signer)
1  reserve
2  reserve_liquidity_mint
3  reserve_liquidity_supply
4  reserve_liquidity_fee_receiver
5  reserve_collateral_mint
6  reserve_collateral_supply
7  lending_market
8  lending_market_authority
9  user_transfer_authority (signer)
10 dex_market
11 token_program
12 rent
```

On-chain `process_init_reserve` (`program/src/processor.rs:533`) expects the first slots to be the user liquidity and collateral accounts, followed by Pyth/Switchboard feeds, `clock` , `lending_market_owner` , and other extras before the token program.

```
0  source_liquidity_info
1  destination_collateral_info
2  reserve
3  reserve_liquidity_mint
4  reserve_liquidity_supply
5  reserve_liquidity_fee_receiver
6  reserve_collateral_mint
7  reserve_collateral_supply
8  pyth_product
9  pyth_price
10 switchboard_feed
11 lending_market
12 lending_market_authority
13 lending_market_owner
14 user_transfer_authority (signer)
15 clock
16 rent
17 token_program
--- remaining_accounts (system program、platform_state、pool_ownership、pool_rate_config )
```

Because the SDK omits these accounts and the order diverges entirely, the processor misinterprets every slot (e.g., treats the reserve as the source wallet) and immediately throws `InvalidAccountOwner` , `NotEnoughAccountKeys` , or similar errors, making the instruction unusable in practice.

Recommendations

Align the client account layout with the processor: update the SDK/CLI to pass the full, ordered account list (source/destination token accounts, oracle feeds, `clock` , ownership/ platform PDAs, token program, etc.), and strengthen on-chain validation around account count and ownership. Add integration tests that compose the instruction via SDK to ensure compatibility and catch regressions.

[M-11] `BorrowObligationLiquidity` account needs mismatch processor logic

Severity

Impact: Medium

Likelihood: Medium

Description

After parsing the nine fixed accounts, `process_borrow_obligation_liquidity` treats the remainder as CoalesceFi extras and immediately expects `remaining_accounts[0]` to be the platform config PDA and `remaining_accounts[1]` to be the PoolOwnership PDA (`program/src/processor.rs:2690-2702`).

```
let pool_ownership = get_pool_ownership_at_position(
      remaining_accounts_slice,
      1,
      borrow_reserve_info.key,
      program_id,
  )?; // ⚠️ This ? propagates error if PoolOwnership doesn't exist
```

The helper `get_pool_ownership_at_position` reads that slot and returns an error if the account is missing, out of order, or fails PDA validation; the propagated `?` aborts the transaction.

The legacy Solend SDK (`sdk/src/instruction.rs:1838-1888`) still builds the instruction with only the original nine accounts and never supplies those PDAs.

```
 /// Creates a 'BorrowObligationLiquidity' instruction.
#[allow(clippy::too_many_arguments)]
pub fn borrow_obligation_liquidity(
    program_id: Pubkey,
    liquidity_amount: u64,
    source_liquidity_pubkey: Pubkey,
    destination_liquidity_pubkey: Pubkey,
    borrow_reserve_pubkey: Pubkey,
    borrow_reserve_liquidity_fee_receiver_pubkey: Pubkey,
    obligation_pubkey: Pubkey,
    lending_market_pubkey: Pubkey,
    obligation_owner_pubkey: Pubkey,
    collateral_reserves: Vec<Pubkey>,
    host_fee_receiver_pubkey: Option<Pubkey>,
) -> Instruction {
    let (lending_market_authority_pubkey, _bump_seed) = Pubkey::find_program_address(
        &[&lending_market_pubkey.to_bytes()[..PUBKEY_BYTES]],
        &program_id,
    );
    let mut accounts = vec![
        AccountMeta::new(source_liquidity_pubkey, false),
        AccountMeta::new(destination_liquidity_pubkey, false),
```

```
        AccountMeta::new(borrow_reserve_pubkey, false),
        AccountMeta::new(borrow_reserve_liquidity_fee_receiver_pubkey, false),
        AccountMeta::new(obligation_pubkey, false),
        AccountMeta::new(lending_market_pubkey, false),
        AccountMeta::new_readonly(lending_market_authority_pubkey, false),
        AccountMeta::new_readonly(obligation_owner_pubkey, true),
        AccountMeta::new_readonly(spl_token::id(), false),
    ];
    for collateral_reserve in collateral_reserves {
        accounts.push(AccountMeta::new(collateral_reserve, false));
    }

    if let Some(host_fee_receiver_pubkey) = host_fee_receiver_pubkey {
        accounts.push(AccountMeta::new(host_fee_receiver_pubkey, false));
    }
    Instruction {
        program_id,
        accounts,
        data: LendingInstruction::BorrowObligationLiquidity { liquidity_amount }.pack(),
    }
}
```

Even if callers add the platform config manually, most legacy pools have no pre-existing PoolOwnership account, so the same check fails. As a result, any pool without PoolOwnership—or any client that omits the account—cannot execute the borrow instruction, effectively introducing a denial of service across all such pools.

Recommendations

If PoolOwnership must be mandatory, simultaneously update the SDK, CLI, and migration tooling so every borrow call includes the platform config and PoolOwnership PDAs, and ship utilities to initialize those accounts for legacy pools.

[M-12] `CoalesceFi` instruction discriminants misroute commands

Severity

Impact: Medium

Likelihood: Medium

Description

The `process_instruction` entrypoint only forwards calls to `coalescefi_processor` when the instruction discriminant falls in the 26–38 range (`program/src/processor.rs:415`).

```
pub fn process_instruction<'a>(
    program_id: &Pubkey,
    accounts: &'a [AccountInfo<'a>],
    input: &[u8],
) -> ProgramResult {
```

```
    // CoalesceFi START - Check routing FIRST before any other processing
    if !input.is_empty() {
        let discriminant = input[0];

        // Route CoalesceFi instructions (26-38) immediately
        // These come after Solend's 0-25 range
        // 26: InitPlatformConfig
        // 27-38: Other CoalesceFi instructions
@>        if discriminant >= 26 && discriminant <= 38 {
            return process_coalescefi_instruction(program_id, accounts, input);
        }
    }
```

However, the `CoalesceFiInstruction` enum reuses legacy Solend discriminant values—for example, `InitPlatformConfig` serializes to 20 (`program/src/coalescefi_instruction.rs:156`).

```
    fn discriminant(&self) -> u8 {
        match self {
            Self::InitPlatformConfig => 20, //26
            Self::ModifyUserAccess { .. } => 21, //27
            Self::CreateExclusivePool { .. } => 22,
            Self::UpdatePlatformConfig { .. } => 30, //30
            Self::TransferAdmin { .. } => 31, //31
            Self::CloseAccessControl { .. } => 32,
            Self::InitReserve { .. } => 33,
            Self::RemoveTokenLimit { .. } => 38,
        }
    }
```

When admins or scripts construct these extension instructions through the SDK/CLI, the on-chain program misparses them as `LendingInstruction::ForgiveDebt` and other legacy instructions, so every CoalesceFi management action (whitelists, platform configuration, credit limits, etc.) fails to take effect. Practically, none of the new credit-lending governance flows can reach the main processor, preventing the platform from enforcing or auditing credit borrowing controls.

Recommendations

Align every `CoalesceFiInstruction` discriminant with the 26–38 window expected by `process_instruction` (for example, map `InitPlatformConfig` , `ModifyUserAccess` , etc. sequentially from 26 onward), and add unit or integration tests to confirm that SDK/CLI-built instructions route to `coalescefi_processor` . Also, establish a review gate for future extensions so new discriminants do not collide with existing program instruction ranges.

[M-13] Platform pause checks can be skipped, leaving sensitive ops enabled

Severity

Impact: Medium

Likelihood: Medium

Description

Several handlers aim to honor `PlatformState::is_paused`, yet they only act when callers voluntarily pass the platform-config PDA (`get_platform_state_address`):

```
    let (platform_config_pda, _) = get_platform_state_address(program_id);
    let remaining_accounts = account_info_iter.as_slice();
    for account in remaining_accounts {
@>        if account.key == &platform_config_pda && account.data_len() > 0 {
            let platform_config = PlatformState::unpack(&account.data.borrow())?;
            if platform_config.is_paused {
                return Err(LendingError::InvalidAccountInput.into());
            }
            break;
        }
    }
```

- `process_redeem_reserve_collateral` (`program/src/processor.rs:1560-1570`) looks through `remaining_accounts`; if no match is found, it silently proceeds.
- `process_refresh_obligation` (`program/src/processor.rs:1763-1776`) sets `remaining_start = 1 + 100`, so the search loop never executes for typical account lists—pause is effectively unchecked.
- `process_donate_to_reserve` (`program/src/processor.rs:4470-4479`) mirrors the redeem behavior; omitting the PDA bypasses the pause guard.

Because these functions do not fail closed, adversaries can omit the PDA while the platform is paused and continue withdrawing collateral, refreshing obligations, or donating liquidity—defeating the emergency-stop mechanism.

The logic in `check_platform_pause` function is correct. So `check_platform_pause` function can be used to replace the codes above.

Recommendations

- Require the platform-config PDA as a mandatory account and error out when missing, owned by the wrong program, or unpacking fails like `check_platform_pause` function.
- Alternatively, derive and load the PDA internally so the program is not dependent on user-supplied remaining accounts.

[M-14] `accrue_interest` neglects fixed rate settings

Severity

Impact: Medium

Likelihood: Medium

Description

```
fn _refresh_reserve_interest(
    program_id: &Pubkey,
    reserve_info: &AccountInfo<'_>,
    clock: &Clock,
) -> ProgramResult {
    let mut reserve = Box::new(Reserve::unpack(&reserve_info.data.borrow())?);
    if reserve_info.owner != program_id {
        return Err(LendingError::InvalidAccountOwner.into());
    }

@>    reserve.accrue_interest(clock.slot)?;
    reserve.last_update.update_slot(clock.slot);
    Reserve::pack(*reserve, &mut reserve_info.data.borrow_mut())?;

    Ok(())
}
```

```
 /// Update borrow rate and accrue interest
    pub fn accrue_interest(&mut self, current_slot: Slot) -> ProgramResult {
        // CoalesceFi START - Debug logging for interest accrual tracking
        msg!(
            "accrue_interest: slot={}, last={}",
            current_slot,
            self.last_update.slot
        );
        // CoalesceFi END
        let slots_elapsed = self.last_update.slots_elapsed(current_slot)?;
        if slots_elapsed > 0 {
@>            let current_borrow_rate = self.current_borrow_rate()?;
            let take_rate = Rate::from_percent(self.config.protocol_take_rate);
            self.liquidity
                .compound_interest(current_borrow_rate, slots_elapsed, take_rate)?;
        }
        Ok(())
    }
```

```
pub fn current_borrow_rate(&self) -> Result<Rate, ProgramError> {
        let utilization_rate = self.liquidity.utilization_rate()?;
        let optimal_utilization_rate = Rate::from_percent(self.config.optimal_utilization_rate);
        let max_utilization_rate = Rate::from_percent(self.config.max_utilization_rate);
        if utilization_rate <= optimal_utilization_rate {
            let min_rate = Rate::from_percent(self.config.min_borrow_rate);

            if optimal_utilization_rate == Rate::zero() {
```

```
                return Ok(min_rate);
            }

            let normalized_rate = utilization_rate.try_div(optimal_utilization_rate)?;
            let rate_range = Rate::from_percent(
                self.config
                    .optimal_borrow_rate
                    .checked_sub(self.config.min_borrow_rate)
                    .ok_or(LendingError::MathOverflow)?,
            );

            Ok(normalized_rate.try_mul(rate_range)?.try_add(min_rate)?)
        } else if utilization_rate <= max_utilization_rate {
            let weight = utilization_rate
                .try_sub(optimal_utilization_rate)?
                .try_div(max_utilization_rate.try_sub(optimal_utilization_rate)?)?;

            let optimal_borrow_rate = Rate::from_percent(self.config.optimal_borrow_rate);
            let max_borrow_rate = Rate::from_percent(self.config.max_borrow_rate);
            let rate_range = max_borrow_rate.try_sub(optimal_borrow_rate)?;

            weight.try_mul(rate_range)?.try_add(optimal_borrow_rate)
        } else {
            let weight: Decimal = utilization_rate
                .try_sub(max_utilization_rate)?
                .try_div(Rate::from_percent(
                    100u8
                        .checked_sub(self.config.max_utilization_rate)
                        .ok_or(LendingError::MathOverflow)?,
                ))?
                .into();

            let max_borrow_rate = Rate::from_percent(self.config.max_borrow_rate);
            let super_max_borrow_rate =
Rate::from_percent_u64(self.config.super_max_borrow_rate);
            let rate_range: Decimal = super_max_borrow_rate.try_sub(max_borrow_rate)?.into();

            // if done with just Rates, this computation can overflow. so we temporarily convert
to Decimal
            // and back to Rate
            weight
                .try_mul(rate_range)?
                .try_add(max_borrow_rate.into())?
                .try_into()
        }
    }
```

The `accrue_interest` flow (`sdk/src/state/reserve.rs:344`) still computes the borrow rate via `current_borrow_rate()`, which interpolates between the reserve's min/optimal/max/super_max slopes based on utilization, just like legacy Solend.

Although the codebase introduces `PoolRateConfig::is_fixed()` and `fixed_rate_bps`, and `process_borrow_obligation_liquidity` loads that config, no step wires the fixed rate into interest accrual. Consequently, pools flagged as fixed continue to accrue interest according to utilization dynamics, so the fixed-rate feature is non-functional.

Recommendations

Before accruing interest, detect the fixed-rate flag and substitute the borrow rate—e.g., derive a constant `Rate` from `fixed_rate_bps` or overwrite the reserve's rate parameters—so `liquidity.compound_interest` operates on the intended fixed value. Add coverage for both fixed and variable pools to ensure future updates do not reintroduce the mismatch.

Low findings

[L-01] u64::MAX borrowing disabled by input validation

The `calculate_borrow()` (lines 371-392 in reserve.rs) has special logic for u64::MAX:

```
if amount_to_borrow == u64::MAX {
    // Calculate maximum possible borrow based on:
    // - max_borrow_value (collateral-based limit)
    // - remaining_reserve_borrow (reserve capacity)
    // - available_amount (reserve liquidity)
    let borrow_amount = max_borrow_value
        .try_mul(decimals)?
        .try_div(self.price_upper_bound())?
        .try_div(self.borrow_weight())?
        .min(remaining_reserve_borrow)
        .min(self.liquidity.available_amount.into());
    // ... calculate fees and return result
}
```

However, the input validation (lines 2465-2467) prevents this:

```
const MAX_REASONABLE_AMOUNT: u64 = u64::MAX / 1000; // Leave room for calculations
if liquidity_amount > MAX_REASONABLE_AMOUNT {
    return Err(LendingError::InvalidAmount.into());
}
```

Thus, users cannot use the "borrow maximum available" feature. Remove or modify the input validation to allow u64::MAX.

[L-02] Redundant `can_borrow()` validation in borrow operation

`process_borrow_obligation_liquidity()` has twice `can_borrow()` validations
First can_borrow() check (line 2645):

```
    if !token_limit.can_borrow(liquidity_amount) {
        return Err(LendingError::ExceedsBorrowLimit.into());
    }
```

Second can_borrow() check (line 2890):

```
    if !user_state.can_borrow(&borrow_reserve.liquidity.mint_pubkey, liquidity_amount) {
        return Err(LendingError::ExceedsBorrowLimit.into());
    }
```

Also, validation in add_borrowed() (lines 127-134) has limit validation:

```
    let new_borrowed = limit.borrowed.checked_add(amount).ok_or(ProgramError::Custom(1))?;
    if new_borrowed > limit.limit {
        return Err(ProgramError::InsufficientFunds);
    }
```

Remove the redundant `can_borrow()` calls and let `add_borrowed()` handle the validation.

[L-03] Unreachable auto-whitelist logic in `process_init_reserve()`

`process_init_reserve()` implements two conflicting access control approaches: - Strict whitelist check (lines 881-890): Only pre-whitelisted users can create pools. - Auto-whitelist logic (lines 1188-1224): Automatically whitelist pool creators. However, the auto-whitelist logic is never executed because the function fails at the strict whitelist check before reaching the auto-whitelist code.

```
// Step 1: Strict whitelist check (lines 881-890)
let authorized = is_authorized_for_pool_creation(
    &lending_market,
    user_transfer_authority_info,
    remaining_accounts_slice,
    program_id,
)?;

if !authorized {
    return Err(LendingError::NotWhitelisted.into()); // ← FAILS HERE
}
```

Remove this unreachable auto-whitelist logic.

[L-04] Hardcoded staleness threshold in pyth price

Hardcoded staleness threshold is used to get pyth price in `pyth.rs` Different price feeds need different staleness thresholds, and a staleness check by a hardcoded threshold will be incorrect. Add this threshold to `ReserveConfig` for a different reserve asset.

[L-05] Incorrect platform pause check

The `process_liquidate_obligation_and_redeem_reserve_collateral()` has double platform paused checks, and the first check is incorrect. It only looks in the last 5 accounts (accounts.iter().rev().take(5)), which may not contain the platform config and can be bypassed. The Second Check (Lines 3561-3564) seems correct, but neither checks if the platform config is this program's config. Use `check_platform_pause()` instead of them.

[L-06] Inconsistent token limit constant and lack of validation in `set_access_mode()`

MAX_TRACKED_TOKENS = 10 in constants.rs is defined but never used. It seems to be used to validate max_tokens in set_access_mode(). But in here, there's conflicting constant. - MAX_TRACKED_TOKENS = 10 (unused). - MAX_TOKEN_LIMITS = 5 (actual array size in UserState).

The set_access_mode() function accepts any max_tokens value without validation so if max_tokens > 5, the actual storage array is limited to 5 slots and this causes corruption. Remove `MAX_TRACKED_TOKENS` in constants.rs. Add a validation that max_tokens <= MAX_TOKEN_LIMITS in `set_access_mode()`.

[L-07] Duplicate borrow limit validation in `process_borrow_obligation_liquidity()`

The `process_borrow_obligation_liquidity()` performs identical borrow limit validation twice for token-specific access mode - once at lines 2635-2655 and again at lines 2880-2898, resulting in redundant computation and unnecessary gas costs. Remove the second check (lines 2880-2898) since the first check already validates borrow limits.

[L-08] Inefficient token limit validation in `process_borrow_obligation_liquidity()`

The `process_borrow_obligation_liquidity()` performs redundant account lookups where both `user_access_pda` and `token_limit_pda` resolve to the same account, causing unnecessary computation and code duplication in the token limit checking logic (lines 2635-2655). - Line 2606: user_access_pda = get_user_state_address(program_id, obligation_owner_info.key). - Line 2631-2632: token_limit_pda = get_user_state_address(program_id, obligation_owner_info.key).

Both PDAs are identical! They're derived from the same function with the same parameters. - Lines 2635-2655: The code searches for `token_limit_pda` in remaining accounts and unpacks it as `UserState`. - Lines 2612-2626: The code already found and unpacked the same account as `user_access`.

Thus, the token limit checking should use `user_access` directly.

[L-09] `PoolOwnership` PDA owner not checked

The 'get_pool_ownership_at_position' validates the expected PDA key and data size but does not verify `owner == program_id` before unpacking `PoolOwnership`.

Impact (low): Missing the standard owner check slightly weakens spoof-resistance and diverges from SPL patterns. An attacker cannot create a PDA with the same key, but it is best practice to also check ownership.

Recommendation: Add an owner check before unpacking.

[L-10] `platform_config.pool_reserve` remains none

`PlatformState::pool_reserve` defaults to `None` and no on-chain instruction ever assigns `Some(pubkey)`—a search through `program/src` shows no write to that field.

```
if let Some(existing_reserve) = platform_config.pool_reserve {
    if existing_reserve != *lending_market_info.key {
        return Err(LendingError::InvalidAccountInput.into());
    }
}
```

Consequently, the checks inside `process_init_reserve` (and similar paths) never trigger, so the platform cannot enforce a single bound reserve or detect mismatched configurations. The field is effectively unused and provides no protection today.

Recommendations

Add a concrete write path for the field—e.g., allow admin instructions (`InitPlatformConfig` / `UpdatePlatformConfig`) to set it or persist it after `InitMultiPoolReserve` runs—and cover first-set/re-set/mismatch cases in tests to confirm the guard works. If the binding is out of scope, document or remove the field to avoid a false sense of safety.

[L-11] Init reserve allows AC incomplete pools

In `process_init_reserve` If fewer than 5 remaining accounts are supplied at init, the program completes token inits, transfers initial liquidity, and mints collateral, then returns Ok(()) and skips AC/ownership PDA setup.

Impact: Denial-of-yield. The pool can accept deposits and redeems, but borrowing is blocked because the borrow path requires `PoolOwnership` at the remaining index 1. Additionally, there is no instruction to backfill PoolOwnership, so pools created in this state remain non-borrowable unless re-created.

```
  if remaining_accounts_slice.len() < 5 {
      return Ok(());
  }
```

Recommendations

Return an error when remaining_accounts_slice.len() < 5 during init. Since pool ownership is required.

[L-12] Platform fee receiver update does not affect existing reserves

Docs state that changing the platform `fee_receiver` via `UpdatePlatformConfig` causes subsequent protocol fee transfers to target the new address (Fee Receiver Updates).

In practice, reserves route fees to their own `reserve.config.fee_receiver` and are not automatically updated when `PlatformState.fee_receiver` changes, even when the reserve was initially configured to match the previous platform receiver.

Impact: After an admin updates `PlatformState.fee_receiver`, reserves that were pointing at the old platform receiver continue sending fees to the old account until each reserve is explicitly reconfigured. This contradicts the documentation's expectation that "subsequent protocol fee transfers target the new address." and may result in funds being sent to the wrong sink.

Recommendations

Add a simple boolean at reserve init, e.g. `use_platform_receiver` - If `true`: always use the `PlatformState.fee_receiver` directly.

[L-13] Missing per-reserve access control `PDA` enforces global whitelist

The documentation states, "Optional per-reserve access control piggybacks on the same UserState structure; when that account is absent, the processor falls back to the global whitelist." However, the implementation never derives or creates a `<user,reserve>` PDA.

```
fn process_borrow_obligation_liquidity<'a>(
    program_id: &Pubkey,
    liquidity_amount: u64,
    accounts: &'a [AccountInfo<'a>],
) -> ProgramResult {

    //skip
@> let per_reserve_access = get_access_control_at_position(
        remaining_accounts_slice,
        2,
        obligation_owner_info.key,
        borrow_reserve_info.key,
        program_id,
    )?;
}
```

```
fn get_access_control_at_position<'a>(
    remaining_accounts: &'a [AccountInfo<'a>],
    position: usize,
    expected_user: &Pubkey,
    _expected_reserve: &Pubkey,
    program_id: &Pubkey,
) -> Result<Option<UserState>, ProgramError> {
@>    let (expected_pda, _) = get_user_state_address(program_id, expected_user);

    let result = get_account_at_position::<UserState>(
        remaining_accounts,
        position,
        program_id,
        &expected_pda,
    )?;

    // Validate the UserState if it exists
    if let Some(ref user_state) = result {
        // Check if UserState is properly initialized
```

```
        if !user_state.is_initialized() {
            return Ok(None);
        }
    }
    Ok(result)
}
```

The borrow flow expects a "per-reserve access" account at `remaining_accounts[2]` (`program/src/processor.rs:2713`), yet `get_access_control_at_position` (`program/src/processor.rs:167`) derives the address via `get_user_state_address(program_id, expected_user)` , ignoring the `_expected_reserve` parameter and therefore pointing back to the global `UserState` .

```
fn process_modify_user_access<'a>(
    program_id: &Pubkey,
    accounts: &'a [AccountInfo<'a>],
    user: Pubkey,
    operation: AccessOperation,
) -> ProgramResult {
    let iter = &mut accounts.iter();
    let admin = next_account_info(iter)?;
    let user_info = next_account_info(iter)?;
    let system = next_account_info(iter)?;
    let platform_state_info = next_account_info(iter)?;

    validate_signer(admin)?;

    // Validate admin authorization
    let (platform_pda, _) = get_platform_state_address(program_id);
    validate_pda(platform_state_info, &platform_pda)?;

    if platform_state_info.owner != program_id {
        return Err(LendingError::InvalidAccountOwner.into());
    }

    let platform_state = PlatformState::unpack(&platform_state_info.data.borrow())?;
    if platform_state.admin != *admin.key {
        return Err(LendingError::InvalidAccountInput.into());
    }

    let (pda, bump) = get_user_state_address(program_id, &user);

    let account_just_created = user_info.data_is_empty();
    if account_just_created {
        create_pda_account(
            program_id,
            admin,
            user_info,
            system,
            UserState::LEN,
@>            &[COALESCEFI_SEED, b"user_state", user.as_ref(), &[bump]],
        )?;
    }

    let mut state = if account_just_created {
        // Initialize new UserState
```

```
        UserState::new(user)
    } else {
        // Unpack existing UserState
        UserState::unpack(&user_info.data.borrow())?
    };
    match operation {
        AccessOperation::SetAccess {
            whitelisted,
            blacklisted,
        } => state.set_access(whitelisted, blacklisted),
        AccessOperation::SetMode { mode, max_tokens } => state.set_access_mode(mode,
max_tokens),
        AccessOperation::SetTokenLimit { mint, limit } => {
            if state.get_token_limit(&mint).is_none() {
                state.add_token_limit(mint, limit)?;
            } else if let Some(tl) = state.get_token_limit_mut(&mint) {
                tl.limit = limit;
            }
        }
        AccessOperation::UpdateTokenLimit { mint, new_limit } => {
            state
                .get_token_limit_mut(&mint)
                .ok_or(LendingError::TokenLimitNotFound)?
                .limit = new_limit;
        }
    }

    state.pack_into_slice(&mut user_info.data.borrow_mut());
    Ok(())
}
```

```
pub fn get_user_state_address(program_id: &Pubkey, user: &Pubkey) -> (Pubkey, u8) {
    Pubkey::find_program_address(
        &[
            COALESCEFI_SEED,
            b"user_state",
            user.as_ref(),
        ],
        program_id,
    )
}
```

The only instruction that creates a `UserState` (`process_modify_user_access` , `program/src/coalescefi_processor.rs:223`) uses the same user-only seeds, and `program/src/coalescefi_state/user_state.rs:352` confirms the PDA derivation omits any reserve component. With no path to create or fetch a reserve-specific account, per-reserve access control never materializes, and the processor cannot fall back gracefully, contradicting the published behavior.

Recommendations

Either implement a genuine per-reserve PDA (for example, `COALESCEFI_SEED + b"user_reserve_access" + user + reserve`) with creation and fallback logic, or update the documentation and runtime checks to reflect that only global whitelist/blacklist state is supported.

[L-14] Borrow flow requires whitelist for every user

Before computing credit, `process_borrow_obligation_liquidity` pulls a `UserState` from `remaining_accounts` and requires `is_whitelisted == true` and not blacklisted (`program/src/processor.rs:2605-2671`).

```
let mut user_is_whitelisted = false;
    {
      //-------------skip

        // If no access control account found, deny access (no backward compatibility)
        if !access_control_checked {
            return Err(LendingError::InvalidAccountInput.into());
        }

        // Double-check whitelist status
@>        if !user_is_whitelisted {
            return Err(LendingError::InvalidAccountInput.into());
        }
    }
```

If the caller omits the account or the user is marked non-whitelisted, the function returns an error, regardless of whether the pool is configured as whitelist-only. Consequently, even pools intended for the public (LTV > 0) become unusable for ordinary borrowers, effectively denying service to every non-whitelisted user.

Recommendations

Separate whitelist enforcement from public borrowing: when the pool policy allows public borrowing, accept missing or non-whitelisted `UserState` and fall back to LTV/collateral checks; only reject when the pool is explicitly whitelist-only or the user is blacklisted. Add regression tests covering both whitelist-required and public pools to guarantee non-whitelisted borrowers can succeed where allowed.

[L-15] `ACCESS_MODE_NO_ACCESS` incorrectly grants borrow privileges

Inside `process_borrow_obligation_liquidity`, the `UserState` access mode is only examined for `ACCESS_MODE_TOKEN_SPECIFIC`; all other modes reuse the earlier `is_whitelisted && !is_blacklisted` result (`program/src/processor.rs:2612-2692`).

```
for (idx, account) in remaining_accounts.iter().enumerate() {
            if account.key == &user_access_pda && account.data_len() > 0 {
                let user_access = UserState::unpack(&account.data.borrow())?;

                // Check if user is blacklisted - immediate denial
                if user_access.is_blacklisted {
                    return Err(LendingError::UserBlacklisted.into());
                }

                // Check if user is whitelisted - REQUIRED for all borrows
```

```
                if !user_access.is_whitelisted {
                    return Err(LendingError::InvalidAccountInput.into());
                }

                user_is_whitelisted = true;

                // For token-specific mode, check and update token limits
@>                if user_access.access_mode == ACCESS_MODE_TOKEN_SPECIFIC {
                    // Look for token limit account
                    let (token_limit_pda, _) =
                        get_user_state_address(program_id, obligation_owner_info.key);

                    let mut token_limit_found = false;
                    for limit_account in remaining_accounts.iter() {
                        if limit_account.key == &token_limit_pda && limit_account.data_len() >
0 {
                            let token_limit_state =
                                UserState::unpack(&limit_account.data.borrow())?;

                            // Get the specific token limit for this mint
                            if let Some(token_limit) = token_limit_state
                                .get_token_limit(&borrow_reserve.liquidity.mint_pubkey)
                            {
                                // Check if amount can be borrowed
                                if !token_limit.can_borrow(liquidity_amount) {
                                    return Err(LendingError::ExceedsBorrowLimit.into());
                                }
                            } else {
                                return Err(LendingError::TokenLimitNotFound.into());
                            }

                            // Note: We'll update the borrowed amount after the borrow succeeds
                            token_limit_found = true;
                            break;
                        }
                    }

                    if !token_limit_found {
                        return Err(LendingError::TokenLimitNotFound.into());
                    }
                }

                access_control_checked = true;
                break;
            }
        }
```

The subsequent `AccessControl::has_access` check also ignores the access mode. Consequently, a user marked `ACCESS_MODE_NO_ACCESS` but left whitelisted still passes both checks and may borrow, contradicting the intended "NO_ACCESS blocks borrowing" policy.

Recommendations

Explicitly reject `ACCESS_MODE_NO_ACCESS` during validation—either by failing the instruction when that mode is detected or by automatically clearing `is_whitelisted` whenever the mode is set.