Coalesce Security Review

Intro

This security review was conducted by `kriko.eth`. The subject was the `Coalesce` Solana program at commit `0d45878`.

Engagement Summary

The subject of this review was the `Coalesce` Solana program, a fixed-term, fixed-rate USDC lending protocol at commit `0d45878`. Issues identified during the initial review were addressed by the Coalesce team in commit `0307aeb`.

A mitigation review of `0307aeb` identified additional concerns, which were addressed in commit `113f9de`.

A final verification of `113f9de` identified minor refinements, addressed in commit `cefd78a`.

Risk Classification

The risk is assessed based on the severity matrix:

Likelihood		Impact	
	HIGH	MEDIUM	LOW
HIGH	**CRITICAL**	**HIGH**	**MEDIUM**
MEDIUM	**HIGH**	**MEDIUM**	**LOW**
LOW	**MEDIUM**	**LOW**	**INFO**

Findings Summary

Total issues discovered:

Severity	Count
Critical	3
High	3
Medium	2
Low	2
Info	4

Critical severity findings

COAL-C01: Fee reservation in settlement factor computation causes lender haircuts and can permanently deadlock markets

Summary

The protocol claims lender withdrawals take priority over fee collection, but the settlement factor computation in `withdraw.rs` and `re_settle.rs` reserves accrued protocol fees from the vault balance before computing the amount available to lenders, contradicting that ordering. In the common case this produces irreversible lender haircuts on every distressed market with nonzero fees. In the extreme case, when accrued fees meet or exceed the vault balance, the settlement factor clamps to 1, every lender payout computes to zero, and every instruction that could modify the deadlocked state is blocked by its own guards, permanently locking all funds with no on-chain recovery path.

Finding Description

Root cause: fee-first reservation in settlement computation. Fees accrue in `accrue_interest` in `src/logic/interest.rs` on virtual interest growth. In a distressed market, these virtual fee claims are unbacked by actual tokens.

The settlement logic in `process` in `src/processor/withdraw.rs` treats fees as senior claims:

```
// withdraw.rs:186-197, fees reserved BEFORE lender availability computed
let vault_balance = u128::from(vault_token.amount());
let fees_reserved = {
    let fees = u128::from(market.accrued_protocol_fees());
    if vault_balance < fees {
        vault_balance    // ALL vault funds reserved as fees
    } else {
        fees
    }
};
let available_for_lenders = vault_balance
    .checked_sub(fees_reserved)
    .ok_or(LendingError::MathOverflow)?;
```

The settlement factor is then computed from `available_for_lenders`, not from `vault_balance`. The identical pattern is replicated in `process` in `src/processor/re_settle.rs`.

This contradicts the guard in `process` in `src/processor/collect_fees.rs`, which explicitly enforces fee subordination:

```
// SR-057: Block fee collection during market distress
let settlement_factor = market.settlement_factor_wad();
if settlement_factor > 0 && settlement_factor < WAD {
    return Err(LendingError::FeeCollectionDuringDistress.into());
}
```

The guard blocks fee extraction during distress but does not prevent the reservation from haircutting lenders first.

Scenario 1: lender haircut (fees < vault_balance). At realistic parameters (10M USDC, 10% APR, 5% fee, 1yr), the base lender loss from fee reservation alone is ~$52,578 per market regardless of repayment level (higher on-chain due to the fee inflation amplifier described below). At aggressive parameters (50% APR, 5% fee, 5yr, 30% repay), the entire vault can be consumed.

Scenario 2: total deadlock (fees >= vault_balance). When `accrued_protocol_fees >= vault_balance`, `available_for_lenders` is zero. The settlement factor clamps to a minimum of 1:

```
let capped = if raw > WAD { WAD } else { raw };   // capped = 0
if capped < 1 {
    1    // CLAMPED TO 1
} else {
    capped
}
```

With `settlement_factor = 1` and `WAD = 1e18`, every lender's payout truncates to zero:

```
let payout_u128 = normalized_amount   // e.g. 1,100,000,000,000 (1.1M USDC
↪  raw)
    .checked_mul(settlement_factor)    // * 1
    .ok_or(LendingError::MathOverflow)?
    .checked_div(WAD)                  // / 1_000_000_000_000_000_000
    .ok_or(LendingError::MathOverflow)?;  // = 0

if payout == 0 {
    return Err(LendingError::ZeroPayout.into());  // every lender blocked
}
```

Once settlement factor is 1, every instruction that could unblock the market is gated by its own guards: `withdraw` fails on `ZeroPayout`, `collect_fees` is blocked by the distress guard, `re_settle` produces the same settlement factor of 1 (`SettlementNotImproved`), and `withdraw_excess` requires `scaled_total_supply == 0` and `settlement_factor >= WAD`. There is no on-chain recovery path.

Attack scenario (zero-borrow deadlock). A whitelisted borrower creates a market with 33% APR, 5yr maturity, 5% protocol fee, and 1M USDC capacity. Lenders deposit 1M USDC. The borrower never borrows. Fees still accrue because `accrue_interest` in `src/logic/interest.rs` (lines 142-161) computes fee growth on `scaled_total_supply` (total lender deposits), not on `total_borrowed` – so interest and fees accumulate on the virtual entitlement of lenders regardless of whether any tokens left the vault. At maturity, accrued fees exceed the vault balance, settlement produces sf=1, and all 1M USDC is permanently locked. The borrower risked nothing. Other scenarios (severe default in a high-APR market) trigger the same deadlock naturally without any attack.

Impact Explanation

High. In the normal scenario, every lender in every distressed market with nonzero fees takes an irreversible haircut equal to the accrued fee amount. In the extreme scenario, all vault funds are permanently locked with no on-chain recovery. A whitelisted borrower can trigger this intentionally at zero cost.

Likelihood Explanation

High. The haircut scenario fires automatically on every distressed market with nonzero fees, no attacker needed. The deadlock scenario requires fees to exceed the vault balance, reachable at moderate parameters (33% APR, 5yr, 5% fee) and substantially lowered by the fee inflation bug.

Recommendation

Fix 1: remove fee reservation from settlement factor. Consider computing the settlement factor in `src/processor/withdraw.rs` and `src/processor/re_settle.rs` from the full `vault_balance` rather than the fee-reduced balance. Because the settlement factor caps at WAD, healthy markets are unaffected: lenders withdraw exactly their entitlement and fees remain in the vault. In distressed markets, the higher settlement factor correctly prioritizes lenders over fees, matching the protocol's stated design intent. Fee collection timing is already enforced by the existing `collect_fees` distress guard, so no changes are needed there. Evaluate `src/processor/borrow.rs` for the same pattern in pre-maturity borrowable capacity.

Fix 2: add a deadlock escape. Consider allowing lenders to withdraw with zero payout when the settlement factor is too low to produce nonzero payouts (removing the `ZeroPayout` hard-stop so lenders can at least clear their positions), or consider adding an admin instruction to clear `accrued_protocol_fees` per-market as an emergency escape hatch.

Status

Fixed.

COAL-C02: Fee computation uses post-accrual scale factor, systematically over-charging protocol fees and enabling market DoS

Summary

The protocol fee computation in `accrue_interest()` uses `new_scale_factor` (post-accrual) instead of `scale_factor` (pre-accrual). This inflates every fee by the growth multiplier `G/WAD`, creates path-dependent fees that vary with accrual frequency, cascades through settlement to suppress lender payouts and produce false market distress, and at valid extreme parameters overflows `u64` to permanently brick all market operations.

Finding Description

In `accrue_interest` in `src/logic/interest.rs`, the fee computation multiplies scaled supply by `new_scale_factor` instead of the pre-accrual `scale_factor`:

```
let scale_factor = market.scale_factor();                          // line 83:
↪  pre-accrual
let new_scale_factor = mul_wad(scale_factor, total_growth_wad)?; // line
↪  133: post-accrual
```

```
// fee_normalized = scaled_total_supply * scale_factor / WAD * fee_delta_wad
↪  / WAD
// Use new scale_factor for the fee computation (matches spec: fee computed
↪  on current supply at current scale)
let fee_normalized = scaled_total_supply
    .checked_mul(new_scale_factor)       // <-- uses post-accrual scale
    ↪  factor
    .ok_or(LendingError::MathOverflow)?
    .checked_div(WAD)
    .ok_or(LendingError::MathOverflow)?
    .checked_mul(fee_delta_wad)
    .ok_or(LendingError::MathOverflow)?
    .checked_div(WAD)
    .ok_or(LendingError::MathOverflow)?;

let fee_normalized_u64 =
    u64::try_from(fee_normalized).map_err(|_| LendingError::MathOverflow)?;
```

The adjacent comments contradict each other: line 151 documents the formula with `scale_factor`, line 152 directs use of `new_scale_factor`. The code follows the wrong one. The module-level docs confirm intent: "Protocol fees are computed as a fraction of each interest delta", which requires the pre-accrual factor.

Since `new_sf = old_sf * G / WAD`, the overcharge ratio is exactly `G/WAD` – the growth multiplier of the accrual interval. Frequent accruals keep `G` near `WAD` (small overcharge per call); infrequent accruals compound the error. At extreme but valid parameters:

APR	Maturity	Correct Fee	Incorrect Fee	Overcharge
50%	5 years	$558,084	$6,787,228	12.16x
100%	5 years	$7,320,097	$1,078,996,560	147.4x

(Per 1M USDC scaled supply, 5% protocol fee rate, single accrual over full maturity.)

The test helper `fee_delta_after_elapsed` mirrors the incorrect formula, so all fee tests validate the bug against itself.

Market DoS via fee overflow

When `fee_normalized` exceeds `u64::MAX` at the cast on lines 163-164, `MathOverflow` is returned. Since every instruction calls `accrue_interest`, all operations revert permanently. At 100% APR and 100% fee rate (within validation bounds), overflow triggers at ~855M USDC. With the correct formula the same parameters stay well within `u64` range.

```
1. Create market: 100% APR, 5-year maturity, max_total_supply = u64::MAX
2. Attract >= 856M USDC in deposits (each deposit accrues a small interval,
↪  no overflow yet)
3. Borrow all funds; hold off-chain
4. Wait for maturity (expected idle period for fixed-term lending)
5. Any interaction triggers accrue_interest for the full idle period:
   fee_normalized = 856e12 * 147.4 * 146.4 = 1.847e19 > u64::MAX (1.845e19)
↪  => MathOverflow
6. All instructions revert. Lender funds locked. Borrower cannot repay.
7. Admin must set global fee_rate_bps = 0, disrupting every market in the
↪  protocol.
```

False distress and lender-to-fee-authority value transfer

The inflated `accrued_protocol_fees` flows into settlement in `src/processor/withdraw.rs` and `src/processor/re_settle.rs`:

```rust
// withdraw.rs:187-197
let fees_reserved = {
    let fees = u128::from(market.accrued_protocol_fees());
    if vault_balance < fees {
        vault_balance    // ALL vault funds reserved as fees
    } else {
        fees
    }
};
let available_for_lenders = vault_balance
    .checked_sub(fees_reserved)
    .ok_or(LendingError::MathOverflow)?;
```

Inflated `fees_reserved` directly reduces `available_for_lenders`, suppressing the settlement factor. A fully-repaid market gets marked as distressed because the inflated fee reservation consumes what should be lender recovery. After lenders withdraw at the suppressed factor, `re_settle` pushes the factor to `WAD` (since `total_normalized = 0`) and `fee_authority` collects the full inflated amount.

At 50% APR, 5% fee, 5yr, 10M USDC: lenders receive 44.2% of entitlement ($53.7M vs $116M correct) while fee authority captures the $62.3M overcharge.

Impact Explanation

High. Every market with a nonzero fee rate is affected on every accrual. At realistic parameters (10% APR, 1yr), the overcharge is ~10.5% of fees. At extreme but valid parameters, the computation overflows `u64`, permanently bricking all market operations and locking lender funds until protocol-wide admin intervention. The only recovery is setting `fee_rate_bps = 0` on `ProtocolConfig`, which is a global setting — this disrupts fee collection for every market in the protocol, not just the affected one.

Likelihood Explanation

High. Triggers on every accrual where `time_elapsed > 0` and `fee_rate_bps > 0`. The overflow sub-impact requires maximum APR/fee rate with large supply and long idle period, but all parameters pass on-chain validation.

Recommendation

Consider replacing `new_scale_factor` with `scale_factor` in the fee computation in `accrue_interest` in `src/logic/interest.rs`, and removing the misleading comment:

```
         let fee_normalized = scaled_total_supply
-            .checked_mul(new_scale_factor)
+            .checked_mul(scale_factor)
             .ok_or(LendingError::MathOverflow)?
             .checked_div(WAD)
             .ok_or(LendingError::MathOverflow)?
```

Consider also updating the test helper `fee_delta_after_elapsed` to use the pre-accrual scale factor – existing fee tests will correctly fail after the fix.

Status

Fixed.

COAL-C03: PDA pre-funding with rent-exempt minimum permanently blocks protocol operations

Summary

Three code locations use lamport-balance existence checks (`lamports() == 0` or `lamports() > 0`) on deterministic PDAs. Anyone can send 890,880 lamports (~$0.08 at the time of writing) to a target PDA via system transfer, causing the existence check to misfire and permanently blocking the associated instruction. Affected: blacklist checks (deposit, withdraw, borrow, create_market), lender position creation (deposit), and borrower whitelist creation (set_borrower_whitelist). The issue can lead to permanent freezing of existing deposits.

Finding Description

Root cause: Lamport-balance-based existence checks on PDAs whose addresses are publicly derivable. A system transfer pre-funds the PDA while leaving it system-owned with `data_len == 0`. All three sites misinterpret a nonzero lamport balance as proof of program-owned initialization.

Site 1: `check_blacklist` **in** `src/logic/validation.rs`**:**

```
if blacklist_check.lamports() == 0 {
    if protocol_config.is_blacklist_fail_closed() {
        return Err(LendingError::Blacklisted.into());
```

```
    }
    return Ok(());
}
```

Pre-funding causes `lamports() == 0` to be `false`, skipping the early return. The subsequent `owner() != blacklist_program` check fails with `InvalidAccountOwner`. Affects all four callers: deposit, withdraw, borrow, create_market.

Site 2: `process` **in** `src/processor/deposit.rs`**:**

```
let position_exists = lender_position_account.lamports() > 0;
```

Site 3: `process` **in** `src/processor/set_borrower_whitelist.rs`**:**

```
let account_exists = borrower_whitelist_account.lamports() > 0;
```

Sites 2 and 3 share the same bug pattern: pre-funding routes execution into the update branch (which fails on `owned_by(program_id)`), while the create branch, which correctly handles pre-funded accounts via `create_account_with_minimum_balance_signed`, is never reached.

All three blocks are permanent. The program has no code path to reclaim lamports from a system-owned account and no admin rescue instruction.

Impact Explanation

High. Most severely, existing lender funds are permanently locked (blocked withdrawals via blacklist check). Additionally blocks new deposits, borrower whitelisting, borrows, and market creation, all at ~0.00089 SOL per target PDA, mass-computable offline.

Likelihood Explanation

High. Requires no privileges, no special timing, and no victim interaction. All PDA seeds use publicly visible on-chain data. Mass attacks against the entire user base cost dollars.

Recommendation

Consider replacing all three lamport-based existence checks with data-length checks. This matches the safe pattern already used in `create_market.rs:179`. Pre-funded PDAs become usable immediately on first legitimate call with no manual cleanup needed.

1. `src/logic/validation.rs` **line 122:**

```
-    if blacklist_check.lamports() == 0 {
+    if blacklist_check.data_len() == 0 {
         if protocol_config.is_blacklist_fail_closed() {
             return Err(LendingError::Blacklisted.into());
         }
         return Ok(());
     }
```

2. `src/processor/deposit.rs` **line 182:**

```
-    let position_exists = lender_position_account.lamports() > 0;
+    let position_exists = lender_position_account.data_len() > 0;
     if position_exists {
         // Verify ownership before deserializing existing position
         if !lender_position_account.owned_by(program_id) {
```

3. `src/processor/set_borrower_whitelist.rs` **line 79:**

```
-    let account_exists = borrower_whitelist_account.lamports() > 0;
+    let account_exists = borrower_whitelist_account.data_len() > 0;

     if !account_exists && is_whitelisted == 1 {
         // Defense-in-depth: reject if whitelist account already has data
```

Status

Fixed.

High severity findings

COAL-H01: Early lender withdrawals during distress permanently forfeit share of future settlement improvements

Summary

When a market is underfunded at maturity, lenders who withdraw early receive a haircut based on the locked settlement factor. If `re_settle` later improves the factor, only remaining lenders benefit: early withdrawers' shares were already burned. The unclaimed tokens eventually flow to the borrower via `withdraw_excess`.

Finding Description

Settlement factor locks on first withdrawal (`src/processor/withdraw.rs:162-223`)

The first `withdraw` after maturity plus grace period computes and locks the settlement factor into market state. If the vault is underfunded, the factor is below WAD, meaning every lender receives a proportional haircut.

Shares are burned on withdrawal (`src/processor/withdraw.rs:296-307`)

When a lender withdraws, their `scaled_balance` is subtracted from both the position and the market's `scaled_total_supply`. The lender's shares are permanently gone; they cannot participate in any future settlement factor improvement.

`re_settle` recomputes the factor using the reduced `scaled_total_supply` (`src/processor/re_settle.rs:114-135`)

After early withdrawals burn shares, `scaled_total_supply` is smaller. When `re_settle` recomputes the factor, the denominator (`total_normalized`) is reduced, but the vault still holds the haircut tokens that early withdrawers did not receive. This inflates the new factor:

```
// re_settle.rs:114-128 (annotated, not original comments)
let total_normalized = market
    .scaled_total_supply()       // smaller after early withdrawals burned
    ↪ shares
    .checked_mul(scale_factor)
    ...
let raw = available_for_lenders  // vault still holds haircut tokens from
↪ early withdrawers
    .checked_mul(WAD)?
    .checked_div(total_normalized)?;  // smaller denominator → bigger
    ↪ factor
```

The factor can reach WAD even though the vault never received enough to cover all original entitlements.

Scenario: 10M USDC market, two lenders (5M each), ~11.05M total entitlement at maturity:

1. Borrower repays only 6M. Grace period expires. Settlement factor locks at ~0.5377.
2. Lender A withdraws: receives ~2.97M (53.8%). Shares burned, `scaled_total_supply` halves.
3. Borrower repays remaining ~5.1M. `re_settle` called: factor reaches WAD (denominator halved, vault increased).
4. Lender B withdraws: receives ~5.53M (100%). Now `scaled_total_supply` is 0 and settlement factor is WAD, so `withdraw_excess` (which requires both conditions, plus zero accrued fees) unblocks. Borrower extracts ~2.55M – tokens that Lender A was entitled to.

Identical positions, different payouts. No manipulation required.

Impact Explanation

High. Lenders permanently forfeit haircut tokens if the settlement factor later improves. Loss scales linearly with market size and underfunding depth (e.g., ~2.55M on a 10M market at 46% underfunding).

Likelihood Explanation

Medium. Occurs naturally in any distress scenario where at least one lender withdraws before additional repayments arrive.

Recommendation

Consider introducing a cumulative haircut accumulator on the market state that tracks the total normalized gap between what early withdrawers were entitled to and what they actually received.

In `withdraw`, when the settlement factor is below WAD, the difference between the lender's full entitlement and their actual payout would be added to this accumulator after each withdrawal.

In `re_settle`, the accumulator would be subtracted from `available_for_lenders` before recomputing the settlement factor. This prevents tokens already forfeited by early withdrawers from inflating the factor for remaining lenders: factor improvements would only reflect genuinely new funds, not recycled haircut tokens.

Status

Fixed.

COAL-H02: Whitelist borrow capacity bypass via cross-market repayment

Summary

`repay` in `src/processor/repay.rs` checks the repayment amount against the borrower's global `current_borrowed` counter but never checks the target market's outstanding principal. A whitelisted borrower can call `repay` on a market where they never borrowed, resetting their global counter to zero, then borrow again on the original market – exceeding their whitelist cap.

Finding Description

The borrow cap check in `src/processor/borrow.rs` (lines 169-175) relies on a single global counter stored in `BorrowerWhitelist.current_borrowed`:

```
// src/processor/borrow.rs:169-175
let new_wl_borrowed = wl
    .current_borrowed()
    .checked_add(amount)
    .ok_or(LendingError::MathOverflow)?;
if new_wl_borrowed > wl.max_borrow_capacity() {
    return Err(LendingError::GlobalCapacityExceeded.into());
}
```

The repay guard in `src/processor/repay.rs` (lines 168-171) only prevents underflow of that same global counter – it never verifies the repayment amount against the target market's outstanding principal (`total_borrowed - total_repaid + total_interest_repaid`):

```
// src/processor/repay.rs:168-171
let current = wl.current_borrowed();
if amount > current {
    return Err(LendingError::RepaymentExceedsDebt.into());
}
```

Then at lines 136-140, `market.total_repaid` is incremented unconditionally:

```
// src/processor/repay.rs:136-140
let new_total_repaid = market
    .total_repaid()
```

```
    .checked_add(amount)
    .ok_or(LendingError::MathOverflow)?;
market.set_total_repaid(new_total_repaid);
```

There is no check that

```
amount <= market.total_borrowed() - (market.total_repaid() -
↪  market.total_interest_repaid())
```

This means a borrower can direct a repayment at any market they are the designated borrower for, regardless of whether they actually borrowed from it.

Attack scenario (borrower whitelisted for 30,000 USDC, Market A supply cap 45,000):

1. Borrower creates Market B via `create_market` (any whitelisted borrower can create markets). Market B has `total_borrowed = 0`.
2. Borrower borrows 30,000 from Market A. Global `current_borrowed` = 30,000.
3. Borrower calls `repay(30,000)` targeting Market B. The guard passes (`30,000 <= 30,000`). Global `current_borrowed` resets to 0. The 30,000 USDC tokens transfer into Market B's vault; Market B's `total_repaid` becomes 30,000 despite `total_borrowed` being 0.
4. Borrower borrows another 15,000 from Market A using the freed global capacity. Global `current_borrowed` = 15,000.

Result: borrower holds 45,000 USDC (50% over cap). Market A's vault is drained, and lenders face a shortfall at maturity. The 30,000 USDC in Market B's vault is locked (not easily recoverable by the borrower since `withdraw_excess` requires full settlement), so this attack costs the borrower real capital – but a borrower planning to default benefits from extracting more than their cap from Market A.

Impact Explanation

Medium. A whitelisted borrower can exceed their borrow cap by an arbitrary amount, violating the protocol's core risk control. Lenders in the over-borrowed market face losses proportional to the excess.

Likelihood Explanation

High. Requires only a whitelisted borrower who is designated on two or more markets. All steps use standard protocol instructions with no special timing or privileged access beyond the existing whitelist.

Recommendation

Add a per-market outstanding-principal check in `src/processor/repay.rs` before incrementing `total_repaid` (line 136). The market's outstanding principal is

```
total_borrowed - (total_repaid - total_interest_repaid)
```

(since `repay_interest` increments both `total_repaid` and `total_interest_repaid`, subtracting the interest portion isolates principal). Reject with an error if `amount` exceeds this

value:

```
// Insert before line 136 in src/processor/repay.rs
let principal_repaid = market.total_repaid()
    .checked_sub(market.total_interest_repaid())
    .ok_or(LendingError::MathOverflow)?;
let outstanding_principal = market.total_borrowed()
    .checked_sub(principal_repaid)
    .ok_or(LendingError::MathOverflow)?;
if amount > outstanding_principal {
    return Err(LendingError::RepaymentExceedsDebt.into());
}
```

Status

Fixed.

COAL-H03: Blacklisted borrower can withdraw excess funds from matured markets

Summary

`withdraw_excess` in `src/processor/withdraw_excess.rs` performs no blacklist check, allowing a sanctioned borrower to reclaim overpaid funds from a matured market. This violates the protocol's stated security invariant and creates compliance exposure.

Finding Description

Every user-facing instruction that moves funds out of the protocol checks the caller against the external blacklist: `create_market` (line 150), `deposit` (line 113), `borrow` (line 115), and `withdraw` (line 109) all call `check_blacklist()` from `src/logic/validation.rs`.

`withdraw_excess` (`src/processor/withdraw_excess.rs`) does not. It accepts only 7 accounts (line 23) with no `blacklist_check` account, and contains no `check_blacklist()` call.

The funds at stake are the borrower's own overpayment – whatever remains in the vault after all four preconditions are met (market matured, all lenders withdrawn, settlement at WAD, protocol fees collected). Despite being "the borrower's own money," a sanctioned address must not be permitted to move funds through the protocol. Compliance blacklists are enforced at the transfer level, not based on who originally owned the funds.

This contradicts `src/lib.rs` line 30: *"Blacklist checked on all user-facing operations."*

Note: `repay` and `repay_interest` also omit blacklist checks, but those move funds **into** the protocol (borrower pays the vault), which is the opposite direction and does not create the same compliance risk.

Impact Explanation

Medium. A sanctioned borrower can extract funds from the protocol after maturity. Although these are the borrower's own excess funds (not lender or protocol funds), the protocol facilitates a token transfer to a sanctioned address via PDA-signed CPI, which is the compliance violation. The financial exposure is bounded by the excess overpayment amount, not the full vault.

Likelihood Explanation

High. The path is unconditional. Once the four preconditions are met (maturity, all lenders withdrawn, full settlement, fees collected), any borrower – regardless of blacklist status – calls `withdraw_excess` with no additional barriers.

Recommendation

Add a `blacklist_check` account as `accounts[7]` in `withdraw_excess` and call `check_blacklist(blacklist_check, config, borrower.address())?` after the emergency pause check (after line 67). This aligns `withdraw_excess` with the pattern used by `borrow`, `deposit`, `withdraw`, and `create_market`.

Status

Fixed.

Medium severity findings

COAL-M01: Deposit cap tightens over time as interest accrual inflates normalized deposits against fixed `max_total_supply`

Summary

The deposit cap compares the interest-inclusive normalized value of all deposits against the fixed `max_total_supply`. Because `scale_factor` grows with every accrual event, available headroom shrinks monotonically. Late lenders are permanently blocked from depositing even when no new USDC has entered the market.

Finding Description

`process` in `deposit.rs` (lines 137-149) computes `scaled_total_supply * scale_factor / WAD` and compares it against the immutable `max_total_supply` set at market creation. `scale_factor` grows with time, so the normalized value of existing deposits increases while the cap stays fixed.

```
let new_scaled_total = market
    .scaled_total_supply()
    .checked_add(scaled_amount)
    .ok_or(LendingError::MathOverflow)?;
let new_normalized = new_scaled_total
    .checked_mul(scale_factor)          // scale_factor grows over time
```

```
    .ok_or(LendingError::MathOverflow)?
    .checked_div(WAD)
    .ok_or(LendingError::MathOverflow)?;
let max_supply_u128 = u128::from(market.max_total_supply());
if new_normalized > max_supply_u128 {   // fixed cap, growing numerator
    return Err(LendingError::CapExceeded.into());
}
```

A dead `total_deposited` field in `Market` (`market.rs`) tracks raw USDC deposits but is never read, suggesting the developer intended raw-USDC cap semantics:

```
// deposit.rs:251-255: written but never read
let new_total_deposited = market
    .total_deposited()
    .checked_add(amount)
    .ok_or(LendingError::MathOverflow)?;
market.set_total_deposited(new_total_deposited);
```

At 10% APR with a 5% headroom margin (10M deposited, 10.5M cap), interest alone fully consumes the cap in ~6 months with no new deposits.

Impact Explanation

Medium. Every market with `annual_interest_bps > 0` is affected. Deposits are permanently blocked once interest growth consumes the headroom; the borrower has no recourse since `max_total_supply` is immutable.

Likelihood Explanation

Medium. Occurs naturally in every non-zero-rate market once sufficient time passes. No attacker required.

Recommendation

Replace the interest-inclusive cap check in `deposit.rs` (lines 137-149) with a comparison against the raw-USDC `total_deposited` counter that is already maintained. This makes the cap time-invariant. The `scaled_total_supply` update on line 250 must be preserved — only the cap validation logic changes.

Two changes required:

1. `deposit.rs` — **cap check (lines 137-149):** Replace the `new_normalized > max_supply_u128` comparison with `new_total_deposited > max_total_supply`, where `new_total_deposited` is the raw USDC sum already computed on lines 251-254. Move that computation above the cap check so it is available.
2. `withdraw.rs` — **decrement** `total_deposited`**:** `total_deposited` is currently cumulative (incremented on deposit, never decremented). For a net-deposit cap, add a corresponding `market.set_total_deposited(market.total_deposited() - payout)`

in `withdraw.rs` after the transfer (around line 307) so that withdrawals free up cap space for future deposits.

Status

Fixed.

COAL-M02: Abandoned or dust lender positions permanently block borrower excess withdrawal

Summary

A single uncleared lender position permanently blocks the borrower from withdrawing excess funds. The protocol has no force-close, expiration, or sweep mechanism.

Finding Description

`withdraw_excess.rs` requires all lender positions to be cleared before the borrower can reclaim surplus funds:

```
// src/processor/withdraw_excess.rs:105-107
if market.scaled_total_supply() > 0 {
    return Err(LendingError::LendersPendingWithdrawals.into());
}
```

Only `withdraw.rs` decrements `scaled_total_supply`, and it requires the lender's signature. `close_lender_position.rs` also requires the lender's signature and only works when `scaled_balance() == 0` (i.e., after a full withdrawal). `LenderPosition` has no expiration field, and no instruction provides a force-close or admin override.

Dust Griefing ($0.27 per market): `deposit` enforces only `amount > 0` (line 46) and `scaled_amount > 0` (line 132), both of which a 1-unit deposit passes. A 1-unit deposit costs ~$0.27 in PDA rent. The attacker never withdraws. `scaled_total_supply() >= 1` forever, blocking all `withdraw_excess` calls.

In distressed markets, this is irreversible even voluntarily: the dust payout rounds to zero and hits the `ZeroPayout` guard:

```
// src/processor/withdraw.rs:259-261
if payout == 0 {
    return Err(LendingError::ZeroPayout.into());
}
```

The same lockout occurs when a lender is externally blacklisted. `withdraw.rs` calls `check_blacklist` at line 109, which reverts with `Blacklisted` if the lender's PDA is flagged. `close_lender_position` requires `scaled_balance() == 0` (line 120). No party can clear either case.

Impact Explanation

Low. The only blocked operation is `withdraw_excess`, which prevents the borrower from reclaiming surplus funds after all lenders have been paid. No lender funds are at risk.

Likelihood Explanation

High. Requires one `deposit(amount=1)` call with no special access. Lost-wallet and blacklist variants need no attacker at all.

Recommendation

Primary fix (covers all three variants): Add a new `force_close_position` instruction callable by the borrower (or admin) after maturity plus a grace period. This instruction should:

1. Compute the lender's owed payout (same math as `withdraw.rs` lines 245-257).
2. Transfer the payout from the vault into a separate escrow PDA keyed by `[SEED_ESCROW, market, lender]`, so the lender can claim later without a signature or blacklist check.
3. Decrement `scaled_total_supply` by the position's `scaled_balance` and zero out the position.

This unblocks `withdraw_excess` while preserving the lender's funds.

Secondary fix (dust variant only): Enforce a minimum deposit amount in `deposit.rs` (e.g., `if amount < MIN_DEPOSIT { return Err(...) }`). This raises the griefing cost but does not address the lost-wallet or blacklist variants.

Status

Fixed.

Low severity findings

COAL-L01: Mint is not checked to be USDC

Finding Description

Comments and constant names throughout the codebase refer to "USDC", but `create_market` only checks that the provided mint has 6 decimals: it never verifies the actual mint address:

```
// src/processor/create_market.rs
if mint_ref.decimals() != USDC_DECIMALS {
    return Err(LendingError::InvalidMint.into());
}
```

The code enforces a property (decimals) rather than an identity (mint address) despite comments suggesting USDC-only. Any 6-decimal SPL token passes validation. Market creation is gated behind the borrower whitelist, so no fund loss is possible, but a non-USDC market would function correctly while contradicting the protocol's documented intent.

Recommendation

Consider adding the USDC mint pubkey as a constant and validating it in `create_market` alongside the existing decimals check.

Status

Fixed.

COAL-L02: Virtual fee reservation suppresses borrowable liquidity pre-maturity

Finding Description

In `borrow.rs`, borrowable liquidity is computed by subtracting `accrued_protocol_fees()` from the vault balance. However, `collect_fees` blocks fee collection while lenders have deposits, so these reserved fees are virtual — no one can claim them pre-maturity.

```
// borrow.rs:134-138
let vault_balance = vault_token.amount();
let fees_reserved = core::cmp::min(vault_balance,
↪  market.accrued_protocol_fees());
let borrowable = vault_balance
    .checked_sub(fees_reserved)
    .ok_or(LendingError::MathOverflow)?;
```

At aggressive-but-valid parameters (50%+ APR, 5% fee, 4+ year maturity), accrued fees can exceed the vault balance and completely lock out borrowing despite funds sitting in the vault. At conservative parameters suppression is minor (<1.5%), but it silently degrades available liquidity over time with no benefit to any party.

Recommendation

Consider removing the fee subtraction from the borrowable liquidity computation in `src/processor/borrow.rs`. The fee counter in `market.accrued_protocol_fees` is still enforced at settlement, so pre-maturity reservation serves no purpose.

Status

Fixed.

Informational findings

COAL-I01: Overlapping immutable and mutable borrows in instructions

Finding Description

Five admin instructions (`set_admin`, `set_fee_config`, `set_pause`, `set_blacklist_mode`, `set_whitelist_manager`) hold an immutable borrow of `protocol_config_account` while taking a mutable borrow in the same scope. This is undefined behavior under Rust's aliasing rules, though it has no practical effect on current BPF/SBF targets.

`borrow_unchecked()` is bound to a `let` that is still live when `borrow_unchecked_mut()` is called on the same account:

```
let config_data = unsafe { protocol_config_account.borrow_unchecked() };
let config_ref: &ProtocolConfig = bytemuck::try_from_bytes(config_data)...;
// config_data still in scope: overlapping borrow
let config_data_mut = unsafe {
↪  protocol_config_account.borrow_unchecked_mut() };
```

At least one other file (`close_lender_position.rs`) already scopes the immutable borrow inside a `{ }` block before taking the mutable one.

Consider wrapping the immutable borrow in a scoping block so it drops before the mutable borrow, matching the existing pattern in `close_lender_position.rs`. Example for `set_admin.rs` (same change applies to `set_fee_config.rs`, `set_pause.rs`, `set_blacklist_mode.rs`, `set_whitelist_manager.rs`):

Status

Fixed.

COAL-I02: `borrow` instruction does not check `is_whitelisted` flag

Finding Description

In `src/processor/borrow.rs`, the only whitelist gate is the capacity check. The `is_whitelisted` flag on `BorrowerWhitelist` is ignored entirely, unlike `src/processor/create_market.rs` which checks it explicitly. If a whitelist manager sets `is_whitelisted = 0` without also zeroing `max_borrow_capacity` (via `set_borrower_whitelist`), the borrower retains access up to their residual capacity. This requires an inconsistent but possible admin action.

```
// borrow.rs: only capacity is checked, is_whitelisted is never read
if new_wl_borrowed > wl.max_borrow_capacity() {
    return Err(LendingError::GlobalCapacityExceeded.into());
}
```

Consider adding `if wl.is_whitelisted != 1 { return Err(...) }` in `borrow.rs` before the capacity check, consistent with the guard in `create_market.rs`:

Status

Fixed.

COAL-I03: Byte-by-byte instruction data parsing

Finding Description

Multiple processor files parse multi-byte instruction data using individual byte indexing — creating an inconsistency that increases maintenance risk. Multi-byte values are constructed

by listing each byte individually (`data[0], data[1], ..., data[7]`) rather than using slices (e.g., `copy_from_slice(&data[0..16])` as `withdraw.rs` already does for its `u128` field). Note that `withdraw.rs` itself is mixed: it uses slice-based parsing for the `u128` scaled amount but byte-by-byte for its `u64` min_payout field. Adding or removing a field requires manually renumbering every subsequent index.

```
// current pattern (create_market.rs)
let market_nonce = u64::from_le_bytes([
    data[0], data[1], data[2], data[3], data[4], data[5], data[6], data[7],
]);
let annual_interest_bps = u16::from_le_bytes([data[8], data[9]]);
```

Consider using `data[n..n+k].try_into().unwrap()` slices across the affected processor functions.

Status

Fixed.

COAL-I04: Stale discriminator comments in processor files

Finding Description

Six processor files have `/// function_name (disc N)` doc comments where the discriminator number does not match the actual dispatch table in `lib.rs`:

File	Comment	Actual (`lib.rs`)
`deposit.rs`	disc 5	disc 3
`borrow.rs`	disc 6	disc 4
`repay.rs`	disc 7	disc 5
`withdraw.rs`	disc 8	disc 7
`collect_fees.rs`	disc 9	disc 8
`withdraw_excess.rs`	disc 18	disc 11

Because `borrow.rs` incorrectly claims disc 6 and `collect_fees.rs` incorrectly claims disc 9, these wrong comments collide with the correct comments in `repay_interest.rs` (disc 6) and `re_settle.rs` (disc 9), making the overall set of comments appear to have duplicate assignments. Integrators relying on these per-file comments instead of `lib.rs` would build transactions with wrong discriminators.

Consider updating the incorrect doc comments to match `lib.rs`.

Status

Fixed.