HawkProof

SECURITY ASSESSMENT

AggreLend (SVM Rust Contract)

Audit Report

JULY 2025

AggreLend

Prepared by

HawkProof Inc.

Approved by

Li Xiaolan

HawkProof

Table of Contents

Executive Summary

Assessment Overview

HawkProof performed a comprehensive security assessment of AggreLend, a pooled auto-optimizer that allocates liquidity across multiple Solana lending venues to maximize yield. The design consolidates deposits by token mint into a single pool per mint, then routes that pooled liquidity to the currently highest-yield market via cross-program invocations (CPIs) to integrated protocols such as Drift, Kamino, MarginFi, Solend, and DefiTuna. Users interact through two primary operations, deposit and withdraw, while the protocol sponsors periodic optimization transactions that rebalance the entire pool.

The review focused on correctness and safety of user funds under adversarial inputs, robustness against upstream program changes, and the handling of external accounts prior to CPI calls. Strengths include atomicity of user operations, sane custody boundaries, a principled share-index mechanism, and a transaction-level allowlist that materially reduces composability-based attack surface. The most significant risks arise from reading doing overflow multiplication operations, performing unchecked byte slicing, assuming specific deposit ordering in Kamino obligations, and relying on fixed byte offsets for foreign program state.

No direct “funds-steal” path was identified in the normal flow with valid venue accounts. However, several issues can lead to reliable instruction-level denial-of-service (DoS), mis-accounting of pool yield and shares if upstream layouts drift, and brittleness that can surface under unexpected inputs or future venue upgrades. All identified issues have concrete, low-friction remediations. Implementing the recommended owner/length/discriminator checks, correcting the Kamino index assumption, codifying decimals constraints, and replacing unwrap() with explicit error handling will significantly elevate the protocol’s safety margin as total value locked increases.

Protocol Overview

Protocol Details

PROJECT TITLE	GITHUB	PROJECT URL	PLATFORM
AGGRELEND	github.com/aggrelend	aggrelend.com	SOLANA

Core Structure

HawkProof reviewed AggreLend's pooled, auto-optimizing lending aggregator on Solana with a focus on end-user operations (deposit and withdraw). Users deposit supported tokens (USDC, USDT, SOL, WBTC, WETH, etc.) into a PDA user_position account. A single optimization transaction rebalances all positions across the entire pool into integrated markets (Drift, Kamino, MarginFi, Solend, DefiTuna) to chase the best APY. Shares and a pool-level data track proportional ownership of the venue-deployed liquidity.

What's robust:

- Strong CPI guard (check_cpi!) blocks CPI entry and self re-entrancy; whole-transaction allowlist significantly shrinks the flash-loan surface.
- Token custody is clean: deposits require the user's PDA signature; withdrawals require the PDA signer and multiple checks are made.
- Math uses 128-bit arithmetic with scaling headroom; many arithmetic paths use saturating ops.
- User ops are atomic: if a venue CPI fails, the transaction reverts (no partial state writes).

Scope & Methodoloy

Scope

The audit is based on the following specific branches and commit hashes of the codebase repositories:

- AggreLend Anchor Program
- Codebase: https://github.com/aggrelend/aggrelend-anchor
- Commit Hash: 8250edf7ff0d45cf19c7d999e5545a737b6f916c

We listed the files we have audited below:

- programs/aggrelend/src/deposit.rs
- programs/aggrelend/src/withdraw.rs
- programs/aggrelend/src/optimize.rs
- programs/aggrelend/src/create.rs
- programs/aggrelend/src/reset.rs
- programs/aggrelend/src/state.rs
- programs/aggrelend/src/helpers.rs
- programs/aggrelend/src/constants.rs
- programs/aggrelend/src/macros.rs

Methodology

The audit examined the deposit and withdraw instructions and all per-venue helper functions they dispatch to, including the logic that updates the user balance, mints/burns shares, and performs CPIs into integrated venues. Administrative initialization, optimization, and reset instructions were also reviewed. The assessment used manual source review, adversarial input reasoning, and property-based analysis of arithmetic and state transitions. No on-chain tests or privileged data were required; conclusions are based solely on the provided code.

Strengths & Areas For Improvement

Strengths

AggreLend benefits from several strong design choices. All operations are atomic, ensuring no partial state changes if downstream CPIs fail. The share-based accounting model with u128 math and scaled factors provides accuracy and overflow protection under extreme conditions. The privilege model is strict, users cannot move pool funds directly, only via a PDA signer. By allow-listing program IDs, the system reduces risk from arbitrary CPIs, and mint-segregated pools ensure issues in one asset do not affect others. Pre-call checks such as pubkey validation and account refreshes further add reliability.

Areas For Improvement

The withdrawal formula (HKP-AL-01) risks overflow, and Kamino withdraw (HKP-AL-02) assumes the deposit is at index 0, a brittle dependency. Multiple places use unchecked slices and unwraps (HKP-AL-03, HKP-AL-08), enabling trivial panics. Hard-coded byte offsets (HKP-AL-04) and token-program ambiguity (HKP-AL-05, HKP-AL-07, HKP-AL-15) create fragility if upstream layouts or program variants change. Arithmetic assumptions need guards: decimals ≤ 9 are not enforced (HKP-AL-06) and unchecked downcasts (HKP-AL-13) can truncate values. Operational issues include manual account close (HKP-AL-10), hard-coded space allocation (HKP-AL-14), and overloaded errors (HKP-AL-16), which obscure causes and complicate debugging. Addressing these will significantly raise resilience and maintainability.

Security Findings

(ALL FINDINGS HAVE BEEN ADDRESSED AS OF 8/03/2025)

TOTAL FINDINGS: 16

CRITICAL	HIGH	MEDIUM	LOW	SUGGESTIONS
1	2	5	6	2

Summary of Key Issues:

- Insufficient account length validation. Before calling venue CPIs, the program reads raw bytes from external accounts without checking their length. This does not permit direct fund theft—since venue CPIs re-verify inputs—but it does enable reliable instruction-level denial-of-service and creates risk of silent mis-accounting if external layouts drift.
- Unchecked indexing. Multiple paths rely on slicing and unwrap() on attacker-supplied accounts. A maliciously short account can trigger predictable panics and halt execution.
- Kamino withdraw assumption. The withdrawal logic assumes the user's target deposit always resides at index 0 inside the obligation. If this assumption fails, calculations may reference the wrong deposit values.
- Hard-coded byte offsets. Several foreign program state fields are accessed by fixed offsets, making the code brittle to upstream struct changes.
- Missing mint ↔ token program check. A lightweight guard to ensure mints are paired with the correct token program is absent, and should be added.
- Decimals edge case. The calculation 10u128.pow(9 - decimals) underflows if applied to mints with more than nine decimals, which is possible under Token-2022.
- Non-idiomatic account close. User accounts are "closed" by draining lamports rather than using Anchor's close attribute, which is functional but brittle and less maintainable.

Table of Findings

(All Findings Have Been Addressed As Of 8/03/2025)

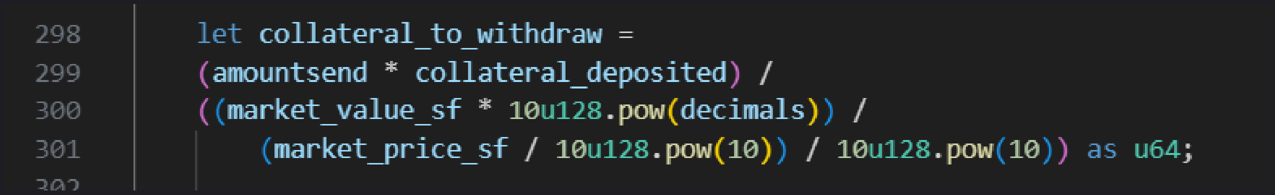
ID	TITLE	SEVERITY	STATUS
HKP-AL-01	Collateral withdrawal calculation overflow risk	CRITICAL	Fixed
HKP-AL-02	Kamino withdraw assumes target deposit at index 0	HIGH	Fixed
HKP-AL-03	Unchecked slicing / unwrap() can panic	HIGH	Fixed
HKP-AL-04	Hard-coded byte offsets of foreign program state	MEDIUM	Fixed
HKP-AL-05	Missing owner/program check for mint ↔ token program pairing	MEDIUM	Fixed
HKP-AL-06	Decimals math pitfalls (10u128.pow(9 - decimals))	MEDIUM	Fixed
HKP-AL-07	Token-program mismatch in Kamino deposit CPI accounts	MEDIUM	Fixed
HKP-AL-08	Missing length checks on remaining_accounts → out-of-bounds panics	MEDIUM	Fixed
HKP-AL-09	Solend path uses permissive fallback when collateral supply is zero	LOW	Fixed
HKP-AL-10	Manual account close via lamport borrow	LOW	Acknowledged
HKP-AL-11	Repeated data.borrow() inside tight loops (micro-inefficiency)	LOW	Fixed
HKP-AL-12	Placeholder oracle accounts in do_refresh_reserve	LOW	Fixed
HKP-AL-13	Unchecked downcasts (as u64, as u8) sprinkled across math paths	LOW	Fixed
HKP-AL-14	Hard-coded space=104 for UserPosition allocation	LOW	Acknowledged
HKP-AL-15	Ambiguous token-program usage for deposit transfer	SUGGESTION	Acknowledged
HKP-AL-16	Overloaded DoNotHavePermission error obscures root causes	SUGGESTION	Fixed

HKP-AL-01 — Collateral withdrawal calculation overflow risk (CRITICAL)

HKP-AL-01	Collateral withdrawal calculation overflow risk	CRITICAL	**Fixed**

Description

In the Kamino withdraw helper, the amount of collateral to withdraw is computed with the following formula:



```rust
let collateral_to_withdraw =
(amountsend * collateral_deposited) /
((market_value_sf * 10u128.pow(decimals)) /
    (market_price_sf / 10u128.pow(10)) / 10u128.pow(10)) as u64;
```

This implementation performs intermediate calculations using u64, including values derived from market_value_sf and market_price_sf. For high-supply, low-price tokens (e.g., BONK-like tokens with small unit price but massive supply), these intermediate values can easily exceed the 64-bit integer range and overflow. Because Rust panics on integer overflow in debug mode (but wraps silently in release mode for unchecked ops), this leads either to program panics or silent corruption of the collateral calculation.

The risk is most acute with tokens that:

- Have low unit prices, yielding extremely large market_price_sf denominators, and
- Have high lamport-denominated balances, magnifying collateral_deposited and amountsend.

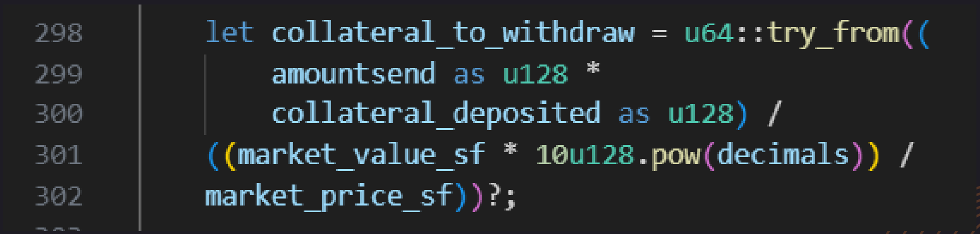
This combination drives intermediate multiplication well beyond 2^64, making u64 insufficient.

Impact:
Overflow Panic: Transaction fails immediately if overflow occurs, preventing user withdrawals.
Silent Mis-Calculation: Depending on compiler optimizations and wrapping semantics, withdrawals could compute a much smaller or larger collateral value than intended.
Custody Divergence: Incorrect collateral calculations can allow under-collateralized withdrawals, diluting pool reserves and directly harming other depositors.

Exploitation Scenario:

- A pool supports BONK (low-price, high-supply token).
- A user deposits a large balance, creating a large collateral_deposited value.
- market_value_sf and market_price_sf reflect BONK's micro-price scale factors, producing very large denominators.
- The multiplication (amountsend * collateral_deposited) produces an intermediate result > 2^64.
- The conversion to u64 silently truncates or panics, yielding:
  - A revert (denial-of-service on withdraws for this pool), or
  - A mis-calculated collateral_to_withdraw that causes the withdrawing user to extract more collateral than originally planned.

Fixed Code Recommendation
All calculations should be performed entirely in u128, with a final checked conversion to u64 only at the end:



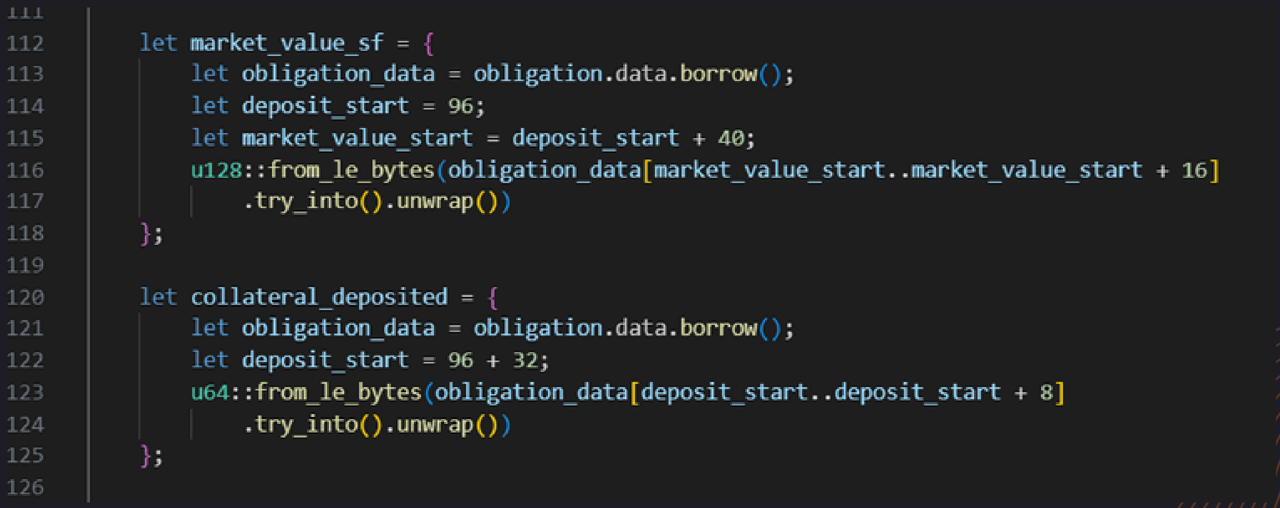
```
    let collateral_to_withdraw = u64::try_from((
        amountsend as u128 *
        collateral_deposited as u128) /
    ((market_value_sf * 10u128.pow(decimals)) /
    market_price_sf))?;
```

HKP-AL-02 — Kamino withdraw assumes target deposit at index 0 (HIGH)

HKP-AL-02	Kamino withdraw assumes target deposit at index 0	HIGH	**Fixed**

Description

withdraw_from_kamino derives market_value_sf and collateral_deposited by reading the first deposit entry in the obligation's deposits array. This assumes the relevant deposit for the pool's mint is always stored at index 0. In contexts where a user's obligation contains multiple deposits, or if ordering changes, the function may compute collateral for the wrong deposit. This can lead to erroneous collateral_to_withdraw and mismatched internal accounting, and in some corner cases could cause the CPI to withdraw an amount for a different asset than the one intended.



```
    let market_value_sf = {
        let obligation_data = obligation.data.borrow();
        let deposit_start = 96;
        let market_value_start = deposit_start + 40;
        u128::from_le_bytes(obligation_data[market_value_start..market_value_start + 16]
            .try_into().unwrap())
    };

    let collateral_deposited = {
        let obligation_data = obligation.data.borrow();
        let deposit_start = 96 + 32;
        u64::from_le_bytes(obligation_data[deposit_start..deposit_start + 8]
            .try_into().unwrap())
    };

```

Impact

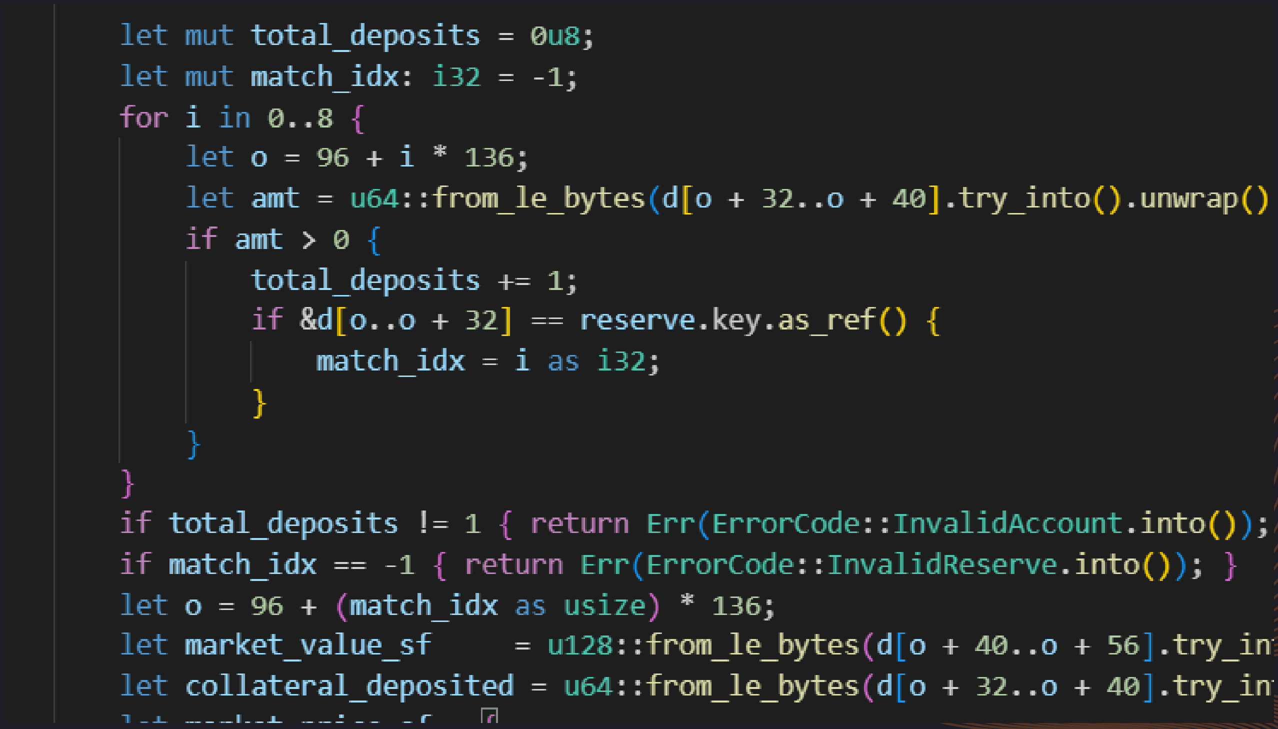
- Incorrect withdrawal math if the target deposit is not at index 0.
- Potential CPI mismatch or partial success that diverges from internal expectations.

Exploitation Scenarios

- Multiple deposits in obligation
  - A user (or adversarial sequence) creates an additional deposit in the same obligation.
  - Index 0 no longer corresponds to the pool's mint.
  - Math uses the wrong deposit values; CPI behavior may not match internal accounting.
- Ordering shift through maintenance or upgrades
  - Upstream venue changes storage order while remaining CPI-compatible.
  - Helper reads index 0 out of habit; accounting diverges.

Recommendations

- Iterate the obligation's deposits and locate the entry whose reserve or mint matches the pool's configured target before reading fields.
- Fail explicitly if the matching entry is not found.
- Apply the same owner/length/discriminator checks described in HKP-01/HKP-02 around all reads.



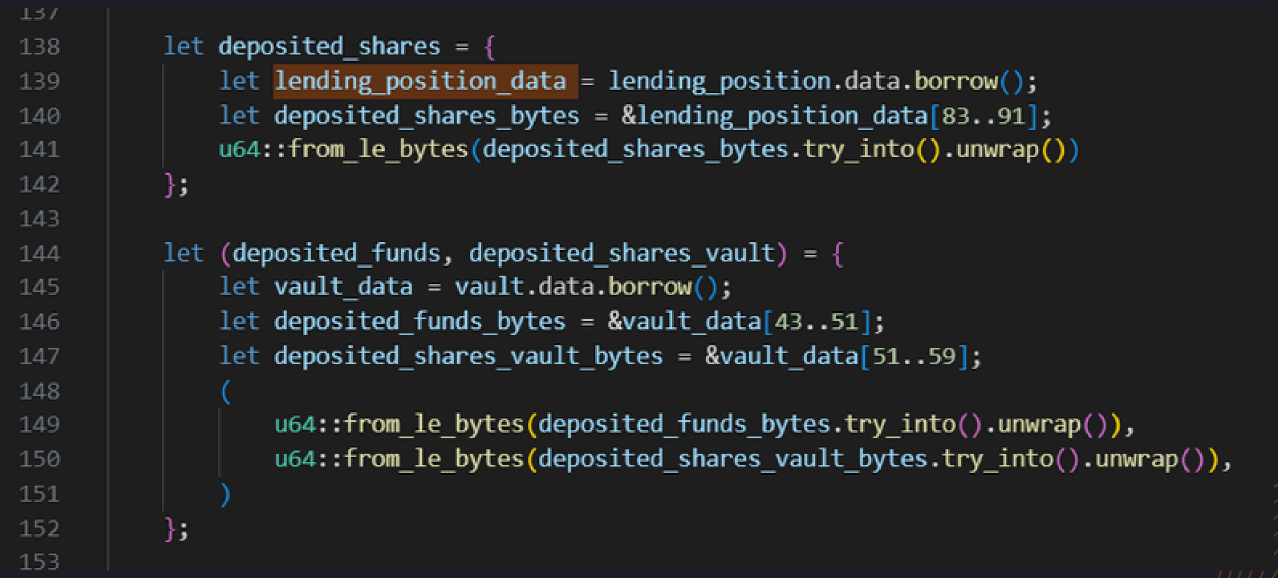
```
    let mut total_deposits = 0u8;
    let mut match_idx: i32 = -1;
    for i in 0..8 {
        let o = 96 + i * 136;
        let amt = u64::from_le_bytes(d[o + 32..o + 40].try_into().unwrap()
        if amt > 0 {
            total_deposits += 1;
            if &d[o..o + 32] == reserve.key.as_ref() {
                match_idx = i as i32;
            }
        }
    }
    if total_deposits != 1 { return Err(ErrorCode::InvalidAccount.into());
    if match_idx == -1 { return Err(ErrorCode::InvalidReserve.into()); }
    let o = 96 + (match_idx as usize) * 136;
    let market_value_sf      = u128::from_le_bytes(d[o + 40..o + 56].try_in
    let collateral_deposited = u64::from_le_bytes(d[o + 32..o + 40].try_in
```

HKP-AL-03— Unchecked slicing / unwrap() can panic (HIGH)

HKP-AL-03	Unchecked slicing / unwrap() can panic	HIGH	**Fixed**

Description

Numerous reads use unchecked byte ranges and try_into().unwrap() conversions on data sourced from accounts provided by the caller. If an account is shorter than expected or not the expected type, the program panics. While a panic reverts the instruction and protects funds, it creates a straightforward path to reliable per-call DoS and produces unhelpful error surfaces for clients.

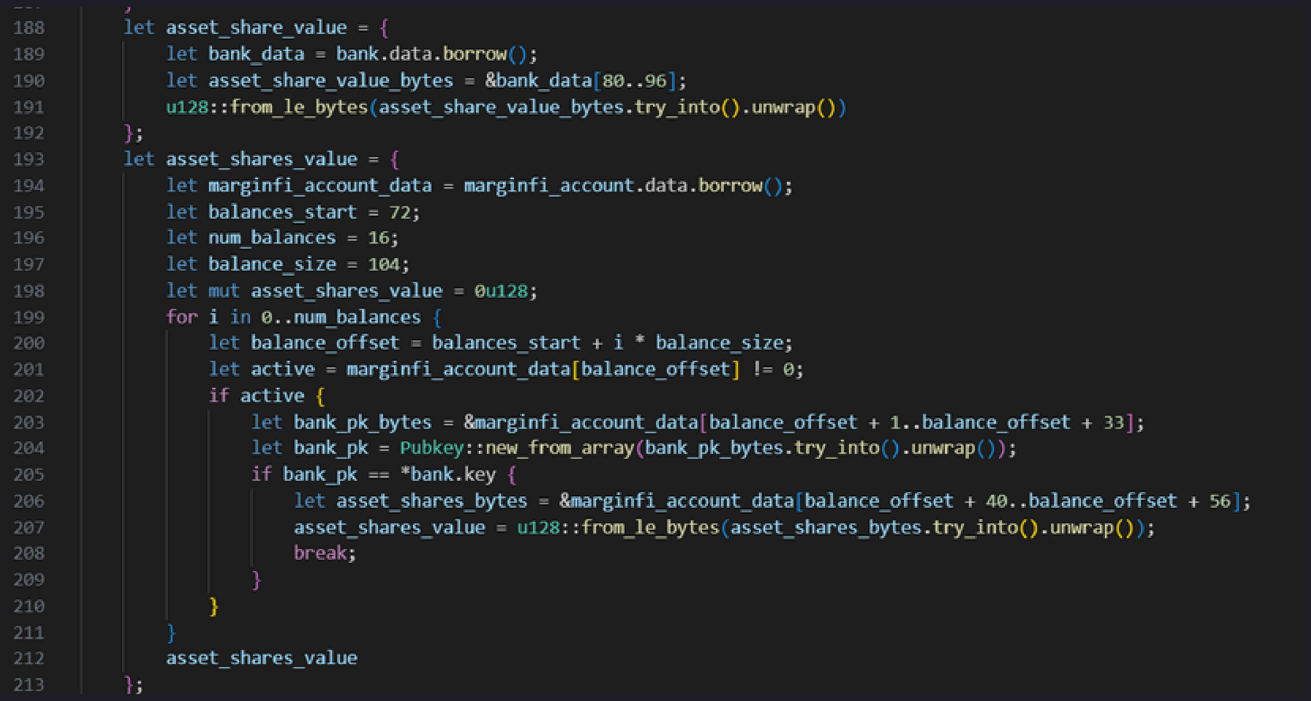


```rust
    let deposited_shares = {
        let lending_position_data = lending_position.data.borrow();
        let deposited_shares_bytes = &lending_position_data[83..91];
        u64::from_le_bytes(deposited_shares_bytes.try_into().unwrap())
    };

    let (deposited_funds, deposited_shares_vault) = {
        let vault_data = vault.data.borrow();
        let deposited_funds_bytes = &vault_data[43..51];
        let deposited_shares_vault_bytes = &vault_data[51..59];
        (
            u64::from_le_bytes(deposited_funds_bytes.try_into().unwrap()),
            u64::from_le_bytes(deposited_shares_vault_bytes.try_into().unwrap()),
        )
    };
```

This implementation performs intermediate calculations using u64, including values derived from market_value_sf and market_price_sf. For high-supply, low-price tokens (e.g., BONK or SHIB-like tokens with small unit price but massive supply), these intermediate values can easily exceed the 64-bit integer range and overflow. Because Rust panics on integer overflow in debug mode (but wraps silently in release mode for unchecked ops), this leads either to program panics or silent corruption of the collateral calculation.

We see the same issue inside the MarginFi calculation:



```
let asset_share_value = {
    let bank_data = bank.data.borrow();
    let asset_share_value_bytes = &bank_data[80..96];
    u128::from_le_bytes(asset_share_value_bytes.try_into().unwrap())
};
let asset_shares_value = {
    let marginfi_account_data = marginfi_account.data.borrow();
    let balances_start = 72;
    let num_balances = 16;
    let balance_size = 104;
    let mut asset_shares_value = 0u128;
    for i in 0..num_balances {
        let balance_offset = balances_start + i * balance_size;
        let active = marginfi_account_data[balance_offset] != 0;
        if active {
            let bank_pk_bytes = &marginfi_account_data[balance_offset + 1..balance_offset + 33];
            let bank_pk = Pubkey::new_from_array(bank_pk_bytes.try_into().unwrap());
            if bank_pk == *bank.key {
                let asset_shares_bytes = &marginfi_account_data[balance_offset + 40..balance_offset + 56];
                asset_shares_value = u128::from_le_bytes(asset_shares_bytes.try_into().unwrap());
                break;
            }
        }
    }
    asset_shares_value
};
```

Impact

- Reliable instruction-level DoS and confusing runtime errors for users.
- Increased compute usage on error paths and noisier preflight outcomes.

Exploitation Scenario

- Supply a 64-byte account for a field assumed at offset 248.
- The first attempted slice beyond the buffer panics; the instruction fails deterministically.

Recommendations

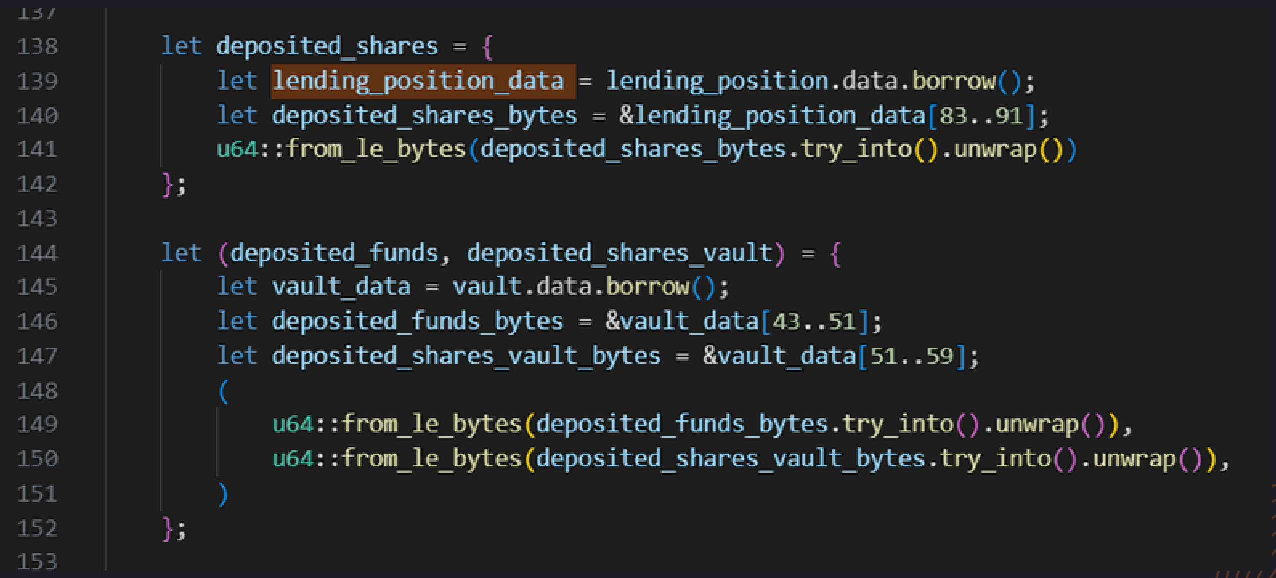
- Adopt safe read utilities that check range bounds and convert with error mapping instead of panicking.
- Normalize all pre-CPI reads to return ErrorCode::InvalidAccountData rather than causing a runtime panic.

HKP-AL-04 — Hard-coded byte offsets of foreign program state

HKP-AL-04	Hard-coded byte offsets of foreign program state	MEDIUM	**Fixed**

Description

All venue helpers interpret external account data via hard-coded offsets. While many protocols append fields rather than reorder them, changes in packing, padding, or versioning can invalidate offsets. Since your internal accounting occurs before CPI, it can silently diverge from reality even when custody operations remain correct.



```
    let deposited_shares = {
        let lending_position_data = lending_position.data.borrow();
        let deposited_shares_bytes = &lending_position_data[83..91];
        u64::from_le_bytes(deposited_shares_bytes.try_into().unwrap())
    };

    let (deposited_funds, deposited_shares_vault) = {
        let vault_data = vault.data.borrow();
        let deposited_funds_bytes = &vault_data[43..51];
        let deposited_shares_vault_bytes = &vault_data[51..59];
        (
            u64::from_le_bytes(deposited_funds_bytes.try_into().unwrap()),
            u64::from_le_bytes(deposited_shares_vault_bytes.try_into().unwrap()),
        )
    };
```

Affected Code (representative)

- Solend reserve/obligation fields at offsets 171, 179, etc.
- Drift cumulative deposit interest at [8..][456..+16].
- MarginFi bank asset_share_value at [80..96].
- Kamino obligation value fields starting at 96, 96+40, etc.

Impact

- Silent mis-accounting of yield and shares following upstream layout changes.
- Unexpected failures if new layouts become smaller than assumed.

Exploitation Scenario

1. Venue introduces a version byte or padding that shifts fields; CPI calls still succeed.
2. Helper continues to read stale offsets and misprices shares across depositors.

Recommendations

- For Anchor accounts, verify discriminator and minimum length and consider checking a known version or size for the structs you depend upon.
- When available, prefer view CPIs or stable accessor routines over raw offset parsing.
- Maintain an allowlist of exact reserve/market pubkeys per pool to constrain ambiguity.

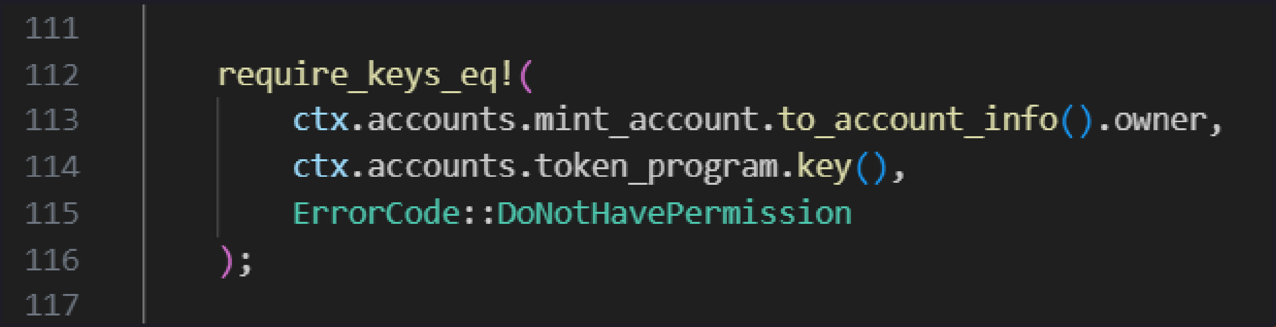
HKP-AL-05 — Missing owner/program check for mint ↔ token program pairing

HKP-AL-05	Missing owner/program check for mint ↔ token program pairing	MEDIUM	**Fixed**

Description

The program accepts a mint_account: InterfaceAccount<Mint> and token_program: Interface<TokenInterface> but does not assert that the mint's owner equals the provided token program's pubkey. While downstream ATAs and venue CPIs usually enforce correct pairing, an early validation removes ambiguity and yields cleaner errors.

Affected Code
deposit / withdraw accounts do not contain:



```rust

    require_keys_eq!(
        ctx.accounts.mint_account.to_account_info().owner,
        ctx.accounts.token_program.key(),
        ErrorCode::DoNotHavePermission
    );

```

Impact

- No direct funds risk, but increases the chance of confusing failures and wasted compute if mint/program mismatch is supplied.

Recommendations

- Add the above assertion to both deposit and withdraw to codify the assumption and fail early with a clear error.
- Optionally persist the chosen token program per pool and assert consistency on later calls.

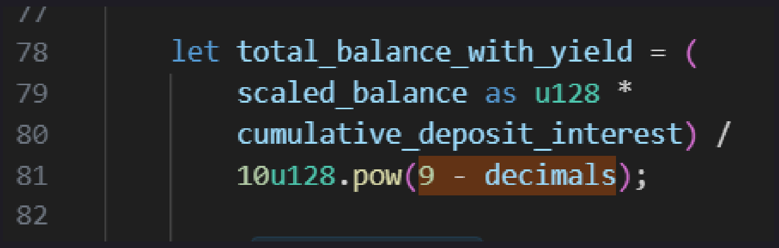
HKP-AL-06 — Decimals math pitfalls (10u128.pow(9 - decimals))

HKP-AL-06	Decimals math pitfalls (10u128.pow(9 - decimals))	MEDIUM	**Fixed**

Description

Some calculations derive a divisor as 10u128.pow(9 - decimals). If a mint with more than nine decimals is ever introduced (possible with both token 2022 & main token program), this expression underflows and panics. The current configuration lists a carefully selected set of mints where decimals are at or below nine, so the issue is latent today but should be guarded explicitly.

Affected Code (example):



```
    let total_balance_with_yield = (
        scaled_balance as u128 *
        cumulative_deposit_interest) /
        10u128.pow(9 - decimals);
```

Impact

- Underflow panic if used with decimals > 9. (possible with both token 2022 & main token program)
- Latent reliability bug dependent on future configuration.

Recommendations

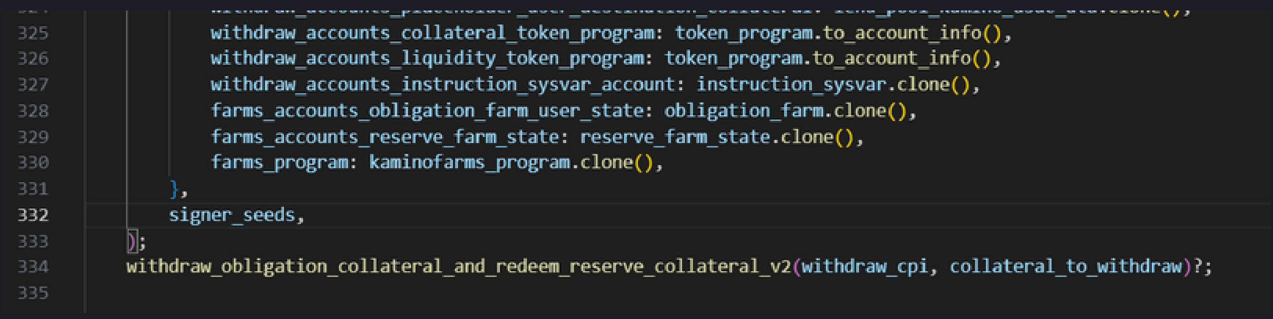
- Enforce require!(decimals <= 9, ErrorCode::UnsupportedToken); after MINT_TABLE lookup, or generalize the math for decimals > 9 by multiplying the numerator by 10^(decimals - 9) instead of dividing.

HKP-AL-07 — Token-program mismatch in Kamino deposit CPI accounts

HKP-AL-07	Token-program mismatch in Kamino deposit CPI accounts	MEDIUM	**Fixed**

Description

In deposit_into_kamino, the CPI uses the same token program two times: both token_program for collateral_token_program and token_program for liquidity_token_program. If the collateral mint happens to be Token-2022 (PYUSD) and default_token_program is classic (or vice-versa), the CPI can fail or behave unexpectedly.



```
                withdraw_accounts_collateral_token_program: token_program.to_account_info(),
                withdraw_accounts_liquidity_token_program: token_program.to_account_info(),
                withdraw_accounts_instruction_sysvar_account: instruction_sysvar.clone(),
                farms_accounts_obligation_farm_user_state: obligation_farm.clone(),
                farms_accounts_reserve_farm_state: reserve_farm_state.clone(),
                farms_program: kaminofarms_program.clone(),
            },
            signer_seeds,
        );
        withdraw_obligation_collateral_and_redeem_reserve_collateral_v2(withdraw_cpi, collateral_to_withdraw)?;
```

Impact

- Configuration foot-gun: subtle mismatches cause venue CPI rejection.
- Increases operational complexity when supporting both token programs.

Example Scenario

1. Liquidity mint is Token-2022; token_program points to classic SPL Token.
2. CPI fails with a generic program error.
3. Operations halt until the correct token program is threaded.

Recommendations

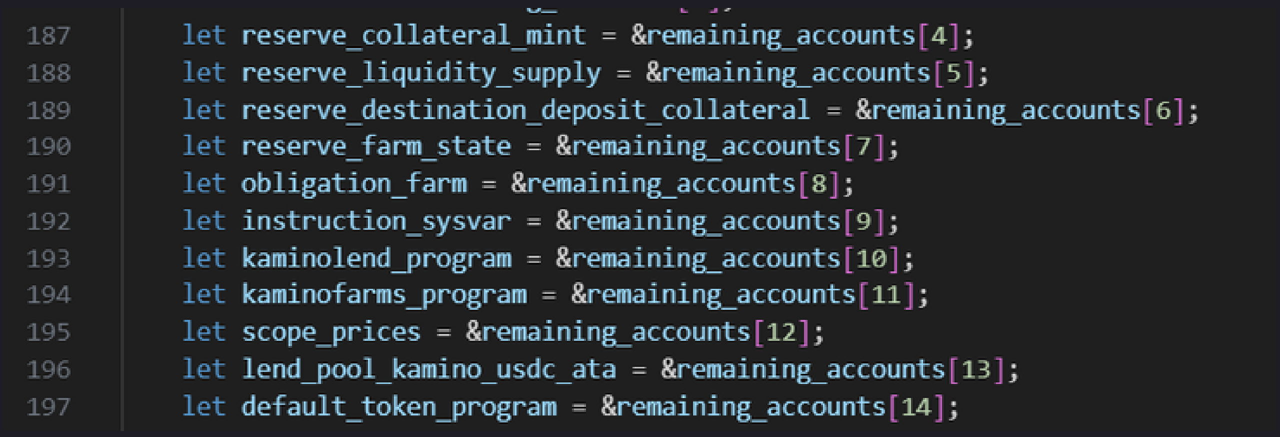
- Derive both token programs from the actual owners of the respective mints (collateral and liquidity), or require them explicitly as accounts and assert ownership.
- Add an event/log line stating the chosen program ids for transparency.

HKP-AL-08 — Missing length checks on remaining_accounts → out-of-bounds panics

HKP-AL-08	Missing length checks on remaining_accounts → out-of-bounds panics	MEDIUM	**Fixed**

Description

Many paths index remaining_accounts directly (e.g., rem[14], remaining_accounts[9]) without verifying the vector length first. If a caller supplies fewer accounts than expected, the program will panic with an out-of-bounds access before reaching any require! guard.



```
    let reserve_collateral_mint = &remaining_accounts[4];
    let reserve_liquidity_supply = &remaining_accounts[5];
    let reserve_destination_deposit_collateral = &remaining_accounts[6];
    let reserve_farm_state = &remaining_accounts[7];
    let obligation_farm = &remaining_accounts[8];
    let instruction_sysvar = &remaining_accounts[9];
    let kaminolend_program = &remaining_accounts[10];
    let kaminofarms_program = &remaining_accounts[11];
    let scope_prices = &remaining_accounts[12];
    let lend_pool_kamino_usdc_ata = &remaining_accounts[13];
    let default_token_program = &remaining_accounts[14];
```

Impact

- Persistent instruction-level failure by providing too few remaining_accounts.
- Poor diagnosability (generic panic rather than a targeted error).

Example Scenario

1. Caller supplies only 6 remaining accounts for a path that expects ≥ 10.
2. The first index beyond length panics.
3. Repeated calls reliably fail the instruction (no funds loss, but persistent failure).

Recommendations

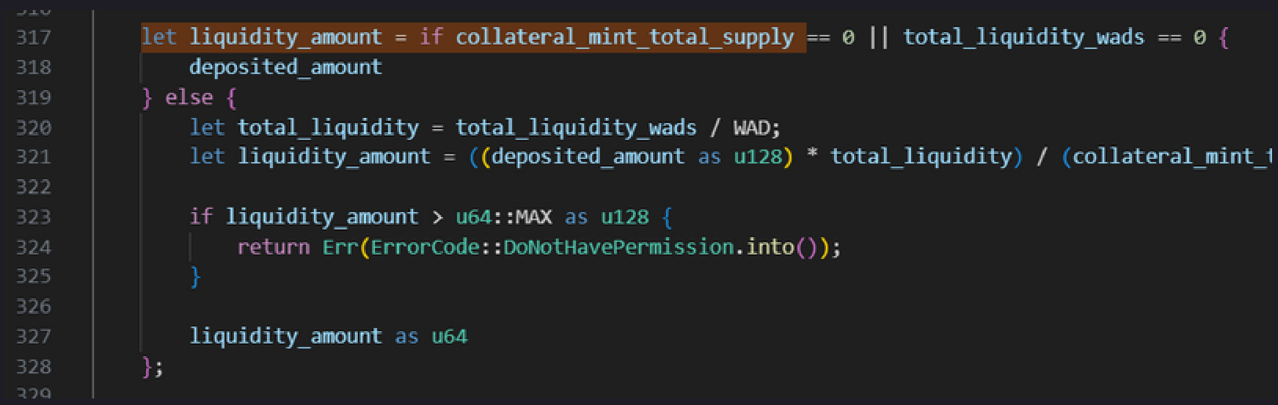
- Add a per-venue constant (const MIN_RA: usize = ...) and require!(rem.len() >= MIN_RA, ErrorCode::InvalidAccountData).
- Prefer de-structuring with early get() + ok_or instead of hard indexes.
- Keep indexes in one place (structured binding) to avoid future drift.

HKP-AL-09 — Solend path uses permissive fallback when collateral supply is zero

HKP-AL-09	Solend path uses permissive fallback when collateral supply is zero	LOW	**Fixed**

Description

In solend_total_balance_with_yield, when collateral_mint_total_supply == 0 (or total_liquidity_wads == 0), the code sets liquidity_amount = deposited_amount as a fallback instead of failing. This may overstate liquidity and propagate incorrect accounting during initialization or edge states.



```
let liquidity_amount = if collateral_mint_total_supply == 0 || total_liquidity_wads == 0 {
    deposited_amount
} else {
    let total_liquidity = total_liquidity_wads / WAD;
    let liquidity_amount = ((deposited_amount as u128) * total_liquidity) / (collateral_mint_t

    if liquidity_amount > u64::MAX as u128 {
        return Err(ErrorCode::DoNotHavePermission.into());
    }

    liquidity_amount as u64
};
```

Impact

- Incorrect balance attribution in rare but important conditions (fresh pools, halted reserves).
- Could enable over-minting of shares or premature withdrawals based on inflated amounts.

Recommendations

- Replace fallback with a hard error (ErrorCode::InvalidLiquidity) and instruct callers to retry after reserve initialization.
- Optionally special-case “brand-new reserve” only when you can prove the right relationship from other fields.

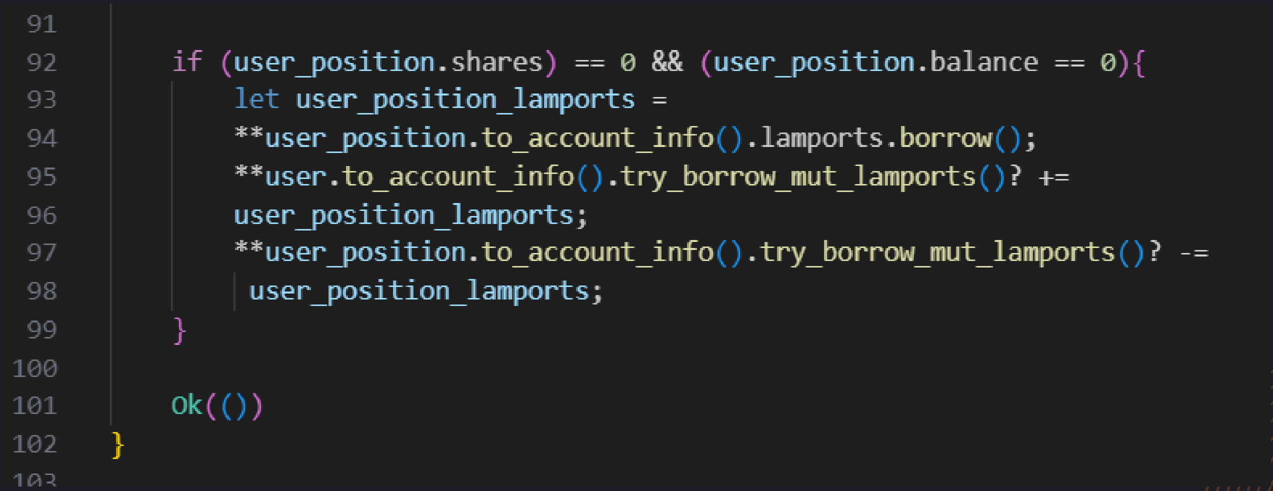
HKP-AL-10 — Manual account close via lamport borrow

HKP-AL-10	Manual account close via lamport borrow	LOW	**Acknowledged**

Description

When a UserPosition is empty, lamports are manually transferred from the account to the user to “close” it, leaving a zero-lamport account until runtime reclamation. This approach is functional and safe when performed only after confirming zero balances, but it is non-idiomatic and can lead to less clear explorer traces and unnecessary edge-case considerations.

Affected Code (withdraw.rs):



```

    if (user_position.shares) == 0 && (user_position.balance == 0){
        let user_position_lamports =
        **user_position.to_account_info().lamports.borrow();
        **user.to_account_info().try_borrow_mut_lamports()? +=
        user_position_lamports;
        **user_position.to_account_info().try_borrow_mut_lamports()? -=
         user_position_lamports;
    }

    Ok(())
}
```

Impact

- No security impact under current gating; mainly a clarity and tooling concern.

Recommendations

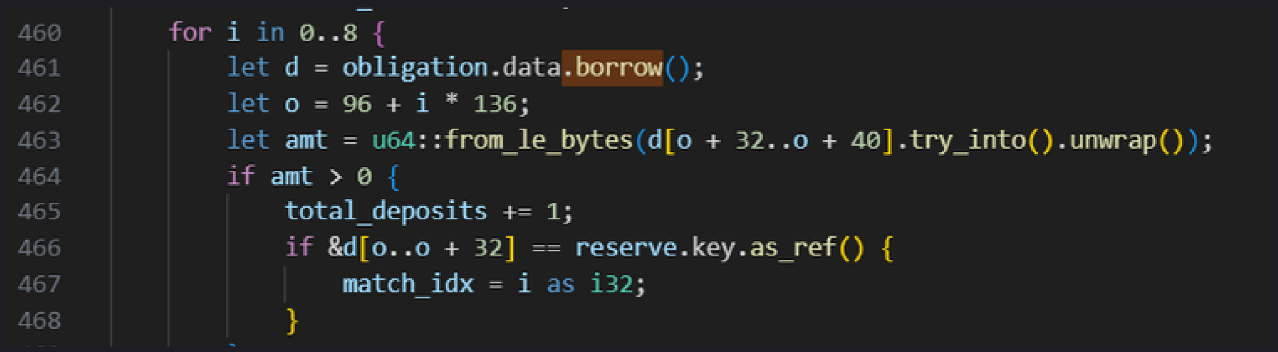
- Prefer the Anchor idiom #[account(close = user)] or assign the account to the system program and realloc(0) before draining to make lifecycle and explorer visibility clearer.

HKP-AL-11 — Repeated data.borrow() inside tight loops (micro-inefficiency)

HKP-AL-11	Repeated data.borrow() inside tight loops (micro-inefficiency)	LOW	**Fixed**

Description

Some loops re-borrow account data each iteration. While the runtime permits it, this creates unnecessary refcount churn and tiny compute overhead.



```
    for i in 0..8 {
        let d = obligation.data.borrow();
        let o = 96 + i * 136;
        let amt = u64::from_le_bytes(d[o + 32..o + 40].try_into().unwrap());
        if amt > 0 {
            total_deposits += 1;
            if &d[o..o + 32] == reserve.key.as_ref() {
                match_idx = i as i32;
            }
```

Impact

- Small, avoidable compute cost and stack activity.
- No security impact.

Example Scenario

1. High-throughput optimizer invokes deposit/withdraw frequently.
2. Repeated borrows add up to measurable CU overhead.
3. Block packing gets marginally tighter after cleanup.

Recommendations

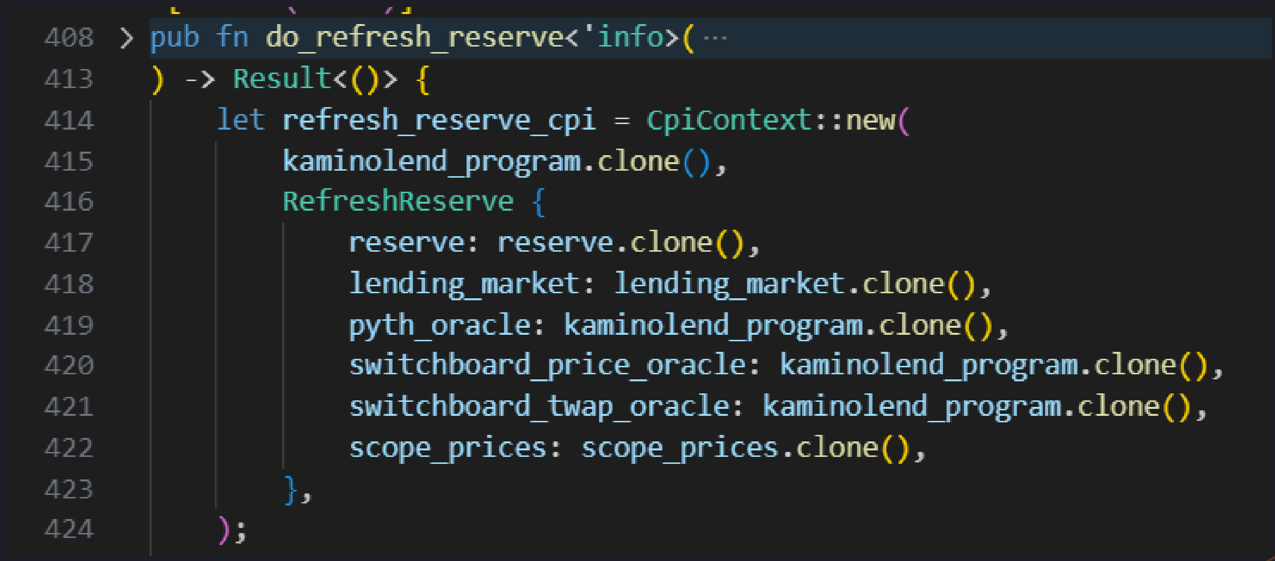
- Borrow once outside the loop (let d = obligation.data.borrow();) and reuse.
- Apply the same pattern to other helpers (MarginFi, Drift, Solend).

HKP-AL-12 — Placeholder oracle accounts in do_refresh_reserve

HKP-AL-12	Placeholder oracle accounts in do_refresh_reserve	LOW	**Fixed**

Description

In Kamino's reserve refresh, oracle accounts (pyth_oracle, switchboard_price_oracle, switchboard_twap_oracle) are supplied as the Kamino program account itself, with scope_prices as the only real market data source. If the venue later requires a real oracle account or validates multiple oracle inputs, this placeholder approach can produce confusing failures.



```rust
pub fn do_refresh_reserve<'info>( ⋯
) -> Result<()> {
    let refresh_reserve_cpi = CpiContext::new(
        kaminolend_program.clone(),
        RefreshReserve {
            reserve: reserve.clone(),
            lending_market: lending_market.clone(),
            pyth_oracle: kaminolend_program.clone(),
            switchboard_price_oracle: kaminolend_program.clone(),
            switchboard_twap_oracle: kaminolend_program.clone(),
            scope_prices: scope_prices.clone(),
        },
    );
```

Example Scenario

1. Kamino starts enforcing that either Pyth or Switchboard accounts be valid.
2. The call passes the program id as a placeholder; refresh fails with a generic error.
3. Operations are blocked until the correct oracles are wired.

Recommendations

- Pass actual oracle accounts where available, or document the dependency on scope_prices clearly.
- Add a feature flag to toggle oracle sources per environment.

HKP-AL-13 — Unchecked downcasts (as u64, as u8) sprinkled across math paths

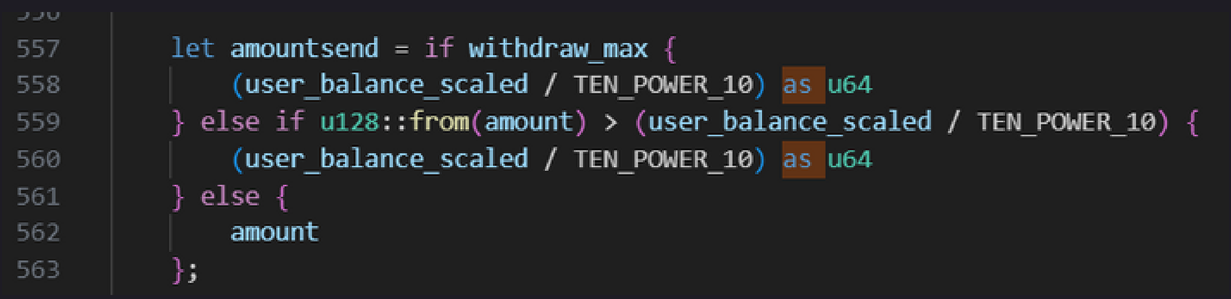
HKP-AL-13	Unchecked downcasts (as u64, as u8) sprinkled across math paths	LOW	**Fixed**

Description

Several code paths cast from wider integers to narrower ones using as, which truncates silently. While many casts are “safe” in normal ranges, they encode assumptions that can bite during venue anomalies.

Affected Code:

- decimals as u8 passed to transfer_checked
- (use_balance_scaled as u64) patterns in helpers
- (liquidity_amount as u64) in Solend math



```rust
    let amountsend = if withdraw_max {
        (user_balance_scaled / TEN_POWER_10) as u64
    } else if u128::from(amount) > (user_balance_scaled / TEN_POWER_10) {
        (user_balance_scaled / TEN_POWER_10) as u64
    } else {
        amount
    };
```

Impact

- Silent truncation on unexpected inputs; harder post-mortems.
- Inconsistent safety posture vs other areas that use try_from.

Example Scenario

A venue scaling change creates a transient large intermediate. A silent downcast trims it, causing small but compounding accounting drift.

Recommendations

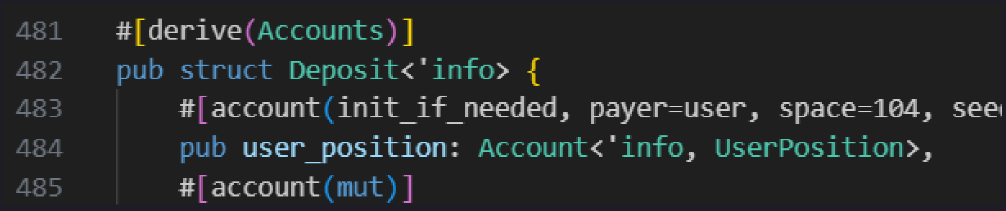
- Use u64::try_from(...)/u8::try_from(...) and bubble a precise error.
- Create helpers: try_u64(v: u128) -> Result<u64> to standardize.

HKP-AL-14 — Hard-coded space=104 for UserPosition allocation

HKP-AL-14	Hard-coded space=104 for UserPosition allocation	LOW	**Acknowledged**

Description

The UserPosition PDA space is set to a literal 104. This matches the current layout (96 bytes + 8-byte Anchor discriminator) but is fragile to future field additions.



```
#[derive(Accounts)]
pub struct Deposit<'info> {
    #[account(init_if_needed, payer=user, space=104, see
    pub user_position: Account<'info, UserPosition>,
    #[account(mut)]
```

Impact

- Future schema changes may cause under-allocation and runtime failures.
- Increases the cost of safe migrations.

Example Scenario

1. A new field is added to UserPosition.
2. Older code paths still allocate 104 bytes.
3. Accounts fail to (re)initialize due to insufficient space.

Recommendations

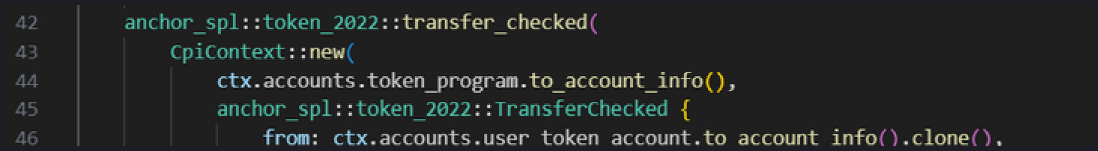
- Use space = 8 + UserPosition::INIT_SPACE or compute with std::mem::size_of::<UserPosition>().
- Introduce a versioned struct pattern if schema evolution is expected.

HKP-AL-15 — Ambiguous token-program usage for deposit transfer

HKP-AL-15	Ambiguous token-program usage for deposit transfer	SUGGESTION	**Acknowledged**

Description

Deposits and withdrawals uses anchor_spl::token_2022::transfer_checked unconditionally while the program accepts a generic TokenInterface for token_program. This is functionally workable when the passed token_program matches the mint's owner, but the intent is not obvious to maintainers and increases the chance of configuration mismatches.



```
anchor_spl::token_2022::transfer_checked(
    CpiContext::new(
        ctx.accounts.token_program.to_account_info(),
        anchor_spl::token_2022::TransferChecked {
            from: ctx.accounts.user_token_account.to_account_info().clone(),
```

Impact

- Maintainability/clarity issue; future contributors may assume token-2022-only behavior.
- Risk of misconfiguration if tooling injects a classic token program for a classic mint without realizing the wrapper in use.

Example Scenario

1. A classic SPL mint (Token v1) is used by ops.
2. Tooling passes the classic token program; the wrapper still compiles but confusion arises about which extra accounts/extensions are expected.
3. Teams lose time debugging an otherwise routine transfer.

Recommendations

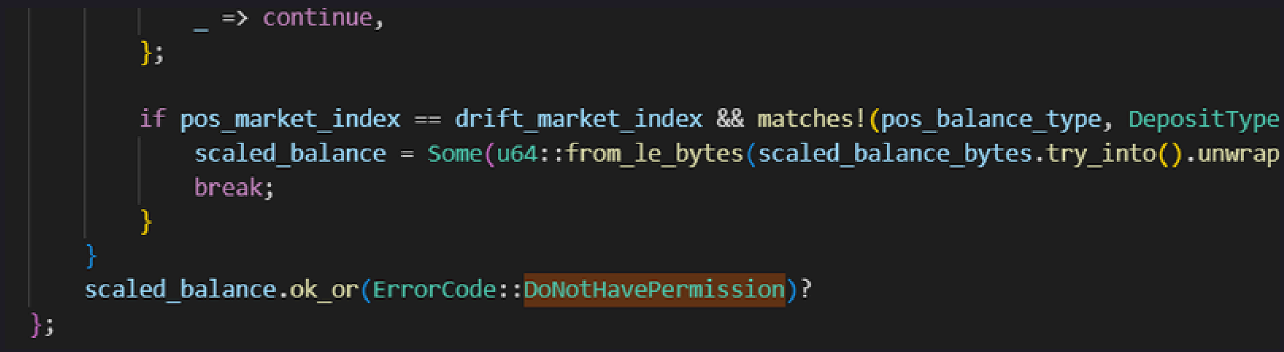
- Branch explicitly on mint.owner:
  - classic: anchor_spl::token::transfer_checked
  - token-2022: anchor_spl::token_2022::transfer_checked
- Or centralize a helper transfer_checked_interface(...) that logs the selected program id for transparency.

HKP-AL-16 — Overloaded DoNotHavePermission error obscures root causes

HKP-AL-16	Overloaded DoNotHavePermission error obscures root causes	SUGGESTION	**Fixed**

Description

The program reuses ErrorCode::DoNotHavePermission for varied validation failures (zero amount, wrong mint, wrong program id, etc.). While functional, overloaded errors degrade UX and telemetry.
Many require! and if branches across deposit_*, *_total_balance_with_yield, and checks such as if amount == 0.



```
            _ => continue,
        };

        if pos_market_index == drift_market_index && matches!(pos_balance_type, DepositType
            scaled_balance = Some(u64::from_le_bytes(scaled_balance_bytes.try_into().unwrap
            break;
        }
    }
    scaled_balance.ok_or(ErrorCode::DoNotHavePermission)?
};
```

Impact

- Harder to distinguish configuration errors from authorization errors.
- Front-end must guess the cause, increasing support burden.

Example Scenario

1. A user passes amount == 0.
2. The program returns DoNotHavePermission.
3. The UI interprets it as an auth issue, confusing the user.

Recommendations

- Add targeted errors: InvalidAmount, WrongMint, WrongProgramId, InvalidMarketIndex, etc.
- Reserve DoNotHavePermission for actual authorization failures.

Recommended Remediation

Remediation (Prioritized)

The first priority is to eliminate correctness flaws that can directly misprice user balances or allow arithmetic failure. Begin with HKP-AL-01 (Collateral withdrawal calculation overflow risk) by moving all intermediate math to u128, introducing explicit checked_mul/checked_div, and using fallible downcasts when converting to u64. This change should be applied uniformly to Kamino, Drift, Solend, and MarginFi paths wherever scaled fixed-point math is performed. In the same tranche, resolve HKP-AL-06 (Decimals math pitfalls) by guarding the 10u128.pow(9 - decimals) pattern and either enforcing decimals ≤ 9 or refactoring the scale so it remains correct for higher-precision Token-2022 mints. These two items remove the most acute sources of overflow and truncation, and they directly protect the integrity of cumulative_yield_index and share calculations.

Next, address venue correctness and account selection logic. HKP-AL-02 (Kamino withdraw assumes target deposit at index 0) should be fixed by locating the user's deposit entry by reserve or mint rather than by position, failing clearly if no match is found, and re-computing market_value_sf/collateral_deposited from the matched entry. In parallel, repair HKP-AL-07 (Token-program mismatch in Kamino deposit CPI accounts) by enforcing that collateral and liquidity token accounts use the correct token program for the specific mint involved; this prevents mixed Token/Token-2022 flows and ensures that the "actual received" amount after CPI can be reconciled. As a venue-edge case, tighten HKP-AL-09 (Solend permissive fallback when collateral supply is zero) by converting the fallback into a hard error so that brand-new or paused reserves cannot propagate misleading balances into the pool.

Remediation (Continued)

With venue-level correctness stabilized, move to input validation and parser hardening. HKP-AL-08 (Missing length checks on remaining_accounts) and HKP-AL-03 (Unchecked slicing / unwrap() can panic) both require a systematic guard layer. Introduce small per-venue wrappers that: (1) assert remaining_accounts.len() meets a minimum; (2) assert each account's owner matches the expected program; (3) assert minimum data lengths (and discriminators for Anchor accounts); and (4) expose named fields so downstream logic never indexes raw slices. This pattern will also reduce the surface area implicated in HKP-AL-04 (Hard-coded byte offsets of foreign program state) by centralizing the offsets in a single, documented location and by adding invariant checks (e.g., version bytes, expected sizes) before any field extraction. Together, these changes convert latent panics into deterministic, domain-specific errors and make future maintenance far safer.

Boundary checks come next. Implement HKP-AL-05 (Missing owner/program check for mint ↔ token program pairing) at the start of both deposit and withdraw so mint.owner == token_program.key() is always enforced, and reconcile HKP-AL-15 (Ambiguous token-program usage for deposit transfer) by applying a single, consistent rule: the token program variable that governs ATA derivation and transfer_checked must be the one that actually owns the mint. This alignment prevents callers from wedging an instruction with inconsistent program choices and ensures transfer semantics (including Token-2022 extensions) are applied predictably across all paths.

Once the safety rails are in place, improve operational clarity and developer ergonomics. HKP-AL-16 (Overloaded DoNotHavePermission error) should be resolved by introducing a more granular error taxonomy—InvalidOwner, InvalidLength, LayoutMismatch, WrongTokenProgramForMint, UnsupportedDecimals, and InvalidReserve—so logs and monitoring can distinguish configuration issues from adversarial input and from venue edge states.

Remediation (Continued)

HKP-AL-10 (Manual account close via lamport borrow) can then be refactored into idiomatic # [account(close = ...)] semantics, which makes intent obvious and future-proofs behavior against runtime changes. Finally, tackle maintainability items: HKP-AL-11 (Repeated data.borrow() inside tight loops) can be addressed by caching slices once per account read; HKP-AL-12 (Placeholder oracle accounts in do_refresh_reserve) should be replaced with real oracle keys and an allowlisted set per network; and HKP-AL-14 (Hard-coded space = 104 for UserPosition) should be replaced by INIT_SPACE derived from the struct or a documented constant computed from field sizes. These last steps do not materially change risk posture but reduce the probability of subtle performance or compatibility issues in future iterations.

To control risk during rollout, apply these changes in staged deployments. Start with a devnet feature flag that enables the new guards and math path; graduate to a canary set of mainnet pools with heightened observability; and only then enable globally. Success criteria for each stage should include: no panics under malformed inputs; exact reconciliation between venue-reported balances and internal accounting; and stable compute usage within established limits.

Defense-in-Depth & Test Plan

Defense Strategies

A durable defense-in-depth posture for AggreLend combines configuration controls, precise erroring, and rigorous verification that mirrors real-world adversarial conditions. Configuration should move away from hard-coded constants toward on-chain, network-scoped registries: store allow-listed program IDs for every integrated venue, token program, and oracle, as well as per-market constants that your parsers depend on. This directly mitigates HKP-AL-12 by ensuring that “placeholder” oracle accounts are rejected at the boundary and gives operations the ability to rotate compromised or deprecated addresses without a code change. A lightweight circuit breaker at the pool level further reduces blast radius during venue incidents: when upstream behavior is anomalous (e.g., prices near zero, reserves reporting impossible totals), new deposits and withdrawals can be paused without touching state, buying time to investigate while preserving user funds.

Verification should begin with property tests over the accounting core. Instrument invariants that capture the spirit of correctness: cumulative_yield_index should be monotonic in the absence of an explicit negative adjustment; shares minted and burned must conserve value under all paths; and “withdraw-max” must map to the exact available entitlement after applying fees and scale factors. These properties directly exercise the fixes for HKP-AL-01 and HKP-AL-06, ensuring that no combination of decimals, prices, or scaled balances can overflow u128, divide by zero, or silently truncate during downcasts as flagged in HKP-AL-13. Complement property tests with differential tests that compare your computed balances against each venue’s canonical method (or a trusted reference implementation) across a matrix of assets, including low-price, high-precision tokens where rounding is most treacherous.

Defense Strategies (Continued)

Input-hardening requires adversarial testing, not just happy-path unit tests. Build fuzzers that generate remaining_accounts vectors of varying lengths, wrong owners, misordered accounts, and under-length data regions to ensure that every path returns targeted, human-readable errors instead of panics—closing the loop on HKP-AL-08 and HKP-AL-03. For HKP-AL-04, maintain a “layout manifest” per venue that lists the byte offsets your parsers consume; write tests that fail loudly when account sizes or version bytes change, forcing a code update rather than permitting silent mis-parsing. For Kamino specifically (HKP-AL-02), synthesize obligations with multiple deposits in different orders, mutate them between refreshes, and assert that your selection logic always resolves the correct entry and never falls back to index-based assumptions.

Observability and operational rehearsal complete the plan. Emit structured events for all critical branches—failed account owner checks, layout mismatches, decimals rejections, and venue selection failures—so dashboards can track error-rate shifts during canary rollouts. Run the full suite under solana-program-test and a local validator with realistic CU limits, then rehearse a canary enablement on a small subset of mainnet pools with pre-defined rollback triggers. Success here is measured not only by correctness under stress but also by clarity: when something fails, the logs should point unambiguously to the failing guard (e.g., WrongTokenProgramForMint vs LayoutMismatch) so remediation is fast and low-friction. By coupling these configuration controls, adversarial test harnesses, and explicit errors, the protocol gains multiple layers of resilience that continue to pay dividends as venues evolve and the asset set grows.

HawkProof

Need an audit or
having any questions?

Contact us.

www.hawkproof.com

team@hawkproof.com

t.me/hawkproof