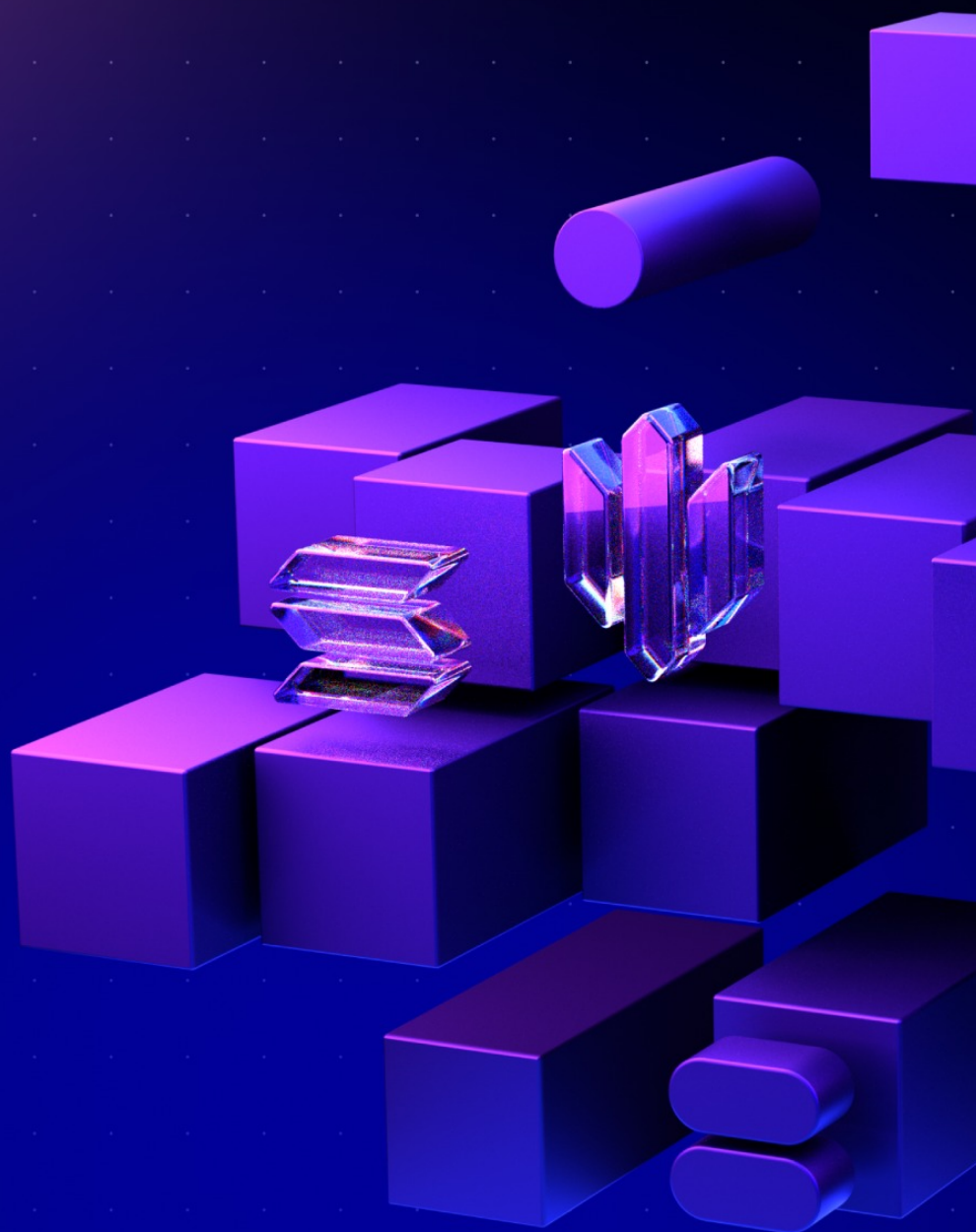
ackee
blockchain security

Project0

DeFi-Native Prime Broker

June 3, 2026



Audit Report

Contents

1. Document Revisions . . . 3
2. Overview . . . 4
   2.1. Ackee Blockchain Security . . . 4
   2.2. Audit Methodology . . . 5
   2.3. Finding Classification . . . 6
   2.4. Review Team . . . 8
   2.5. Disclaimer . . . 8
3. Executive Summary . . . 9
   Revision 1.0 . . . 9
   Revision 1.1 . . . 12
4. Findings Summary . . . 14
Report Revision 1.0 . . . 24
   Revision Team . . . 24
   System Overview . . . 24
   Trust Model . . . 25
   Fuzzing . . . 28
   Findings . . . 30
Report Revision 1.1 . . . 176
   Revision Team . . . 176
Appendix A: How to cite . . . 177
Appendix B: Trident Findings . . . 178
   B.1. Fuzzing . . . 178
Appendix C: Trident Arena Findings . . . 184
   C.1. Discovered Findings . . . 184

1. Document Revisions

1.0	Final Report	May 19, 2026
1.1	Fix Review	June 3, 2026

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**

   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Manual code review**

   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases, project-specific risks, and common classes of Solana program vulnerabilities such as missing ownership checks, signer authorization or rent exemption assertion, signed CPI of unverified programs, cosplay of Solana accounts, bump seed canonicalization, incorrect accounts closing, casting truncation, precision errors, arithmetic overflows or underflows.

3. **Local deployment and hacking**

   Programs are deployed locally, where targeted attempts to exploit vulnerabilities are made. The programs' resilience against various attack vectors is evaluated, with each project's attack attempts being unique to its implementation.

4. **Unit and fuzz testing**

   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Trident framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

5. **Trident Arena assisted vulnerability discovery**

   As the last step, the scope is checked against Trident Arena, an automated vulnerability analysis system for Solana program reviews, to identify potentially missed vulnerabilities. This step is executed at the end of the audit process to avoid distracting auditors from manual review.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Andrej Lukačovič	Lead Auditor
Max Kupchenko	Auditor
Matej Hyčko	Auditor
Artem Delikatnyi	Auditor
Yulian Kisil	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Project0 DeFi-Native Prime Broker is a decentralized margin trading and lending protocol on Solana that enables users to supply assets, borrow against collateral, and access leveraged positions through a risk-engine-driven architecture, with supporting infrastructure for oracle-based pricing, health-factor and liquidation enforcement, isolated/account-level risk controls, on-chain interest rate accrual across supported asset banks, and integrations with external lending venues such as Kamino, JupLend, Drift, Solend, and Staked Collateral pools.

Revision 1.0

Project0 engaged Ackee Blockchain Security to perform a security review of Project0 DeFi-Native Prime Broker with a total time donation of 53 engineering days in a period between February 18 and April 24, 2026, with Andrej Lukačovič as the lead auditor.

The audit was performed on the commit `4fbfaa3`[1] in the project0 repository and the scope was the following:

- marginfi program

Solend integrations were excluded from the scope, namely the following instructions were not reviewed:

- `lending_pool_add_bank_solend`
- `solend_init_obligation`
- `solend_deposit`
- `solend_withdraw`

The audit was split into multiple phases.

In the first phase, we focused on understanding Marginfi's end-to-end architecture, expected account lifecycle, and core risk model assumptions across deposits, borrows, repayments, withdrawals, and liquidations. During this phase we aligned on the intended trust boundaries (protocol admins, liquidators, and external oracle inputs) and clarified the critical accounting and solvency invariants expected to hold throughout the system.

The second phase was dedicated to in-depth manual review of the on-chain logic, with special attention to:

- account constraint correctness (PDA seeds, ownership, signer/writable requirements, and cross-program assumptions);
- solvency and one-way safety invariants across borrow/repay and deposit/withdraw flows;
- correctness of risk and liquidation logic (health computations, thresholds, and edge-case transitions);
- pause / emergency-stop coverage across user-facing and privileged entrypoints;
- administrative/configuration foot-guns and upgrade/IDL mismatches that can break operational controls; and
- safety of integrations with external components in scope (especially oracle-dependent paths).

In parallel, we used Trident to implement fuzz tests for critical accounting and authorization invariants, complementing the manual review with automated adversarial input generation. More details are available in Appendix B.

The third phase focused on consolidating findings, developing minimal proof scenarios where needed to demonstrate impact (especially for accounting, authorization, and liquidation-critical issues), and compiling the audit report

with clear recommendations.

As a final phase, we used Trident Arena as an additional analysis pass to cross-check the review for missed edge cases and consistency issues across the core execution flow. See the Appendix C for the complete list of Trident Arena findings.

Our review resulted in 67 findings, ranging from Info to High severity.

The most severe findings were the following high severity issues.

The H1 is a high severity issue where an account can become ineligible for receivership liquidation or deleverage due to a stale `LiquidationRecord` linkage after account closure and reinitialization.

The H3 is a high severity issue where JupLend-backed orders cannot be executed permissionlessly. As a result, stop-loss and take-profit orders may fail to execute when keepers attempt to trigger them on behalf of offline users.

The H2 is a high severity issue where `juplend_deposit` and `juplend_withdraw` do not invoke the bank or group rate limiter, which can bypass configured circuit breakers during incidents.

The H4 is a high severity issue where permissionless `settle_emissions` can be invoked on disabled accounts, permanently draining `bank.emissions_remaining` while leaving emissions locked and reducing rewards for healthy depositors.

Ackee Blockchain Security recommends Project0:

- read and review the complete audit report; and
- address all identified issues.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

Project0 engaged Ackee Blockchain Security to perform a fix review of the findings from the previous revision.

The review was performed between May 19 and May 31, 2026.

Project0 provided multiple commits that addressed the reported findings.

The issue H3 was fixed in commit `9e8a881`[2] by replacing the bespoke `JuplendWithdraw` constraint with the canonical authorization helpers, enabling keepers to execute JupLend-backed orders permissionlessly.

The issue H1 was not fixed. The client attempted a fix in commit `2e5953f`[3] by introducing a `close_liquid_record` instruction; however, closing still required providing a `marginfi_account` account whose `liquidation_record` matched the existing PDA, so reinitialized accounts remained unable to re-link the stale liquidation record.

The issue H2 was fixed in commit `8f38cfb`[4] by invoking the rate limiter in both `juplend_deposit` and `juplend_withdraw` through the `record_deposit_inflow` and `record_withdrawal_outflow` helpers.

The issue H4 was fixed in commit `f28865d`[5] by removing `lending_account_settle_emissions`, eliminating the permissionless path that could permanently debit `bank.emissions_remaining` on disabled accounts.

For the remaining findings, the relevant fixing commits (where applicable) are referenced in each individual finding description.

From the reported 67 findings:

- 27 issues were fixed;
- 36 issues were acknowledged;

- 3 issues were partially fixed; and
- 1 issue remained unresolved.

No new findings were reported.

See Report Revision 1.1 for the complete fix review.

[1] full commit hash: `4fbfaa3ec13b92fdc3621146edfb786668d2e8e2`, link to commit

[2] full commit hash: `9e8a881eac153fbc50ccf6409d7c1107bcdb7832`, link to commit

[3] full commit hash: `2e5953f8c66732be022985a063cd582ad481f1a3`, link to commit

[4] full commit hash: `8f38cfb9109cbc7ee78cea6fe4c4e9a925933122`, link to commit

[5] full commit hash: `f28865dc05439e9f41a78c3a61cbcc513a06c3f8`, link to commit

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	4	17	24	6	16	**67**

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
H1: Receivership liquidation bypass via dangling LiquidationRecord linkage	**High**	1.0	Fixed
H2: `juplend_deposit` and `juplend_withdraw` do not invoke the rate limiter	**High**	1.0	Fixed
H3: JupLend-backed orders cannot be executed permissionlessly	**High**	1.0	Fixed

Finding title	Severity	Reported	Status
H4: Permissionless `settle_emissions` on `ACCOUNT_DISABLED` balances permanently drains `bank.emissions_remaining`	**High**	1.0	Fixed
M1: Drift withdraw USD valuation double-counts deposit interest	Medium	1.0	Fixed
M2: Permissionless `migrate_curve` can retroactively re-rate unaccrued interest	Medium	1.0	Fixed
M3: Bank creation allows unbounded Token-2022 mint extensions	Medium	1.0	Acknowledged
M4: Permissionless propagation can override manual staked bank configuration	Medium	1.0	Acknowledged
M5: `juplend_withdraw` does not validate remaining-account ordering on `withdraw_all`	Medium	1.0	Fixed
M6: Stop-loss slippage protection degrades when position is underwater	Medium	1.0	Acknowledged
M7: Rate-limit inflow uses cached price while outflow uses live oracle, enabling capacity rental	Medium	1.0	Fixed

Finding title	Severity	Reported	Status
M8: `lending_pool_accrue_bank_interest` is not gated by the protocol pause	Medium	1.0	Acknowledged
M9: `keeper_close_order` / `close_order` rent refund goes to caller-chosen `fee_recipient`	Medium	1.0	Acknowledged
M10: Close-path permanently strands outstanding emissions (both whole units and sub-unit dust)	Medium	1.0	Fixed
M11: Operation-type enforcement checks use tolerance without clamping, allowing sub-threshold invariant violations	Medium	1.0	Fixed
M12: `emissions_destination_account` survives authority transfer — prior owner can drain post-transfer emissions	Medium	1.0	Fixed
M13: Order execution snapshots stale pre-refresh health, enabling value extraction	Medium	1.0	Acknowledged
M14: `juplend_deposit` and `juplend_withdraw` bypass `ACCOUNT_FROZEN` and the canonical authorization pair	Medium	1.0	Fixed

Finding title	Severity	Reported	Status
M15: Stale orders can execute against unrelated positions because `Order` binds to reusable tags rather than stable bank identity	Medium	1.0	Acknowledged
M16: Emissions config changes retroactively reprice already-accrued history	Medium	1.0	Fixed
M17: Outbound vault transfers on Token-2022 transfer-fee mints underpay destinations and misaccount the outstanding counters	Medium	1.0	Partially fixed
L1: Global `panic_pause` does not take effect until per-group propagation	Low	1.0	Acknowledged
L2: Configuring rate limits resets limiter windows, allowing history erasure via no-op updates	Low	1.0	Acknowledged
L3: Emode clone skips configuration validation	Low	1.0	Acknowledged
L4: Isolated assets can be stranded after equity-based bankruptcy disablement	Low	1.0	Acknowledged
L5: `pulse_bank_price_cache` is not gated by the protocol pause	Low	1.0	Acknowledged

Finding title	Severity	Reported	Status
L6: `close_balance` does not update bank total shares or position counts	Low	1.0	Fixed
L7: `transfer_to_new_account` state migration is incomplete — missing field copies, orphan `LiquidationRecord`, and permanently un-closable old account	Low	1.0	Partially fixed
L8: `deposit_up_to_limit` uses pre-accrual capacity, then fails post-accrual	Low	1.0	Fixed
L9: `should_skip_rate_limit` misses `ACCOUNT_IN_ORDER_EXECUTION`	Low	1.0	Acknowledged
L10: `lending_account_withdraw_emissions_permissionless` has no session-flag gate	Low	1.0	Fixed
L11: `marginfi_account_update_emissions_destination_account` does not validate the stored destination's type	Low	1.0	Fixed
L12: Fee state propagation has no version or consistency check between `FeeState` and `group.fee_state_cache`	Low	1.0	Acknowledged

Finding title	Severity	Reported	Status
L13: `lending_pool_withdraw_fees_permissionless` forwards direct vault donations	Low	1.0	Fixed
L14: `juplend_init_position` does not refresh rate or verify fToken mint, yet activates the bank	Low	1.0	Acknowledged
L15: JupLend `claim_account` is unchecked and trusted to be unused by the CPI	Low	1.0	Acknowledged
L16: Tokenless-deleveraging banks still contribute full Initial-margin collateral	Low	1.0	Acknowledged
L17: `lending_account_close_balance` does not check session flags	Low	1.0	Fixed
L18: No bounds validation on fee parameters in `init_global_fee_state` / `edit_global_fee_state`	Low	1.0	Acknowledged
L19: `lending_pool_update_fees_destination_account` has no owner/state check on the destination	Low	1.0	Acknowledged
L20: `lending_pool_withdraw_insurance` accepts unbounded amount, with no accrue or minimum-buffer invariant	Low	1.0	Acknowledged

Finding title	Severity	Reported	Status
L21: Frozen account authorities can still `close_order` and `set_keeper_close_flags`	Low	1.0	Acknowledged
L22: `keeper_close_order` can be bricked by reusing the closed `marginfi_account` address	Low	1.0	Fixed
L23: Shared rate limiter uses unchecked `u64` → `i64` narrowing	Low	1.0	Fixed
L24: Bank onboarding accepts mints whose transfer rules break protocol vault assumptions	Low	1.0	Acknowledged
W1: `daily_limit = 0` disables deleverage withdrawal limit enforcement while setter rejects 0	Warning	1.0	Acknowledged
W2: Group admin can withdraw emissions from frozen accounts to an arbitrary destination	Warning	1.0	Acknowledged
W3: Outflow circuit breakers are controlled by the same admin they intend to constrain	Warning	1.0	Partially fixed
W4: Single-step authority transfer in MarginfiGroupConfigure	Warning	1.0	Acknowledged
W5: New groups ignore global panic state until propagation	Warning	1.0	Acknowledged

Finding title	Severity	Reported	Status
W6: `global_fee_admin` and `global_fee_wallet` rotated in a single-step write with no two-step handover, timelock, or non-zero check	Warning	1.0	Acknowledged
I1: Permissionless bank creation event reports admin as signer	Info	1.0	Fixed
I2: `juplend_init_position` is permissionless and flips `operational_state` to `Operational`	Info	1.0	Acknowledged
I3: `get_asset_shares` silently returns zero when `asset_share_value == 0`	Info	1.0	Acknowledged
I4: `lending_pool_withdraw_fees` has no mint check on `dst_token_account`	Info	1.0	Acknowledged
I5: `update_bank_cache` writes `last_update` independently of `accrue_interest`	Info	1.0	Acknowledged
I6: Several JupLend CPI ancillary accounts are fully unconstrained	Info	1.0	Fixed
I7: `deposit_limit` on JupLend banks is silently denominated in fToken share units	Info	1.0	Fixed
I8: `init_global_fee_state` has no authority check on the payer	Info	1.0	Acknowledged

Finding title	Severity	Reported	Status
I9: JupLend fToken vault seed constant mismatch between type-crate and program	Info	1.0	Fixed
I10: `MarginfiAccount` init and close instructions bypass the protocol pause check	Info	1.0	Fixed
I11: `marginfi_account_init_liq_record` has no authority/`has_one`/pause check	Info	1.0	Acknowledged
I12: `marginfi_account_initialize_pda` — `Some(0)` and `None third_party_id` derive the same PDA	Info	1.0	Acknowledged
I13: `place_order` reuses existing balance tags; `set_keeper_close_flags` has per-balance scope	Info	1.0	Acknowledged
I14: `keeper_close_order` event carries `Pubkey::default()` when `marginfi_account` is already closed	Info	1.0	Fixed
I15: Order-execution `validate_ix_first` rejects allowlisted ATA `Create` / `CreateIdempotent`	Info	1.0	Acknowledged
I16: Several handlers read and act on bank state without first calling `accrue_interest`	Info	1.0	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Andrej Lukačovič	Lead Auditor
Max Kupchenko	Auditor
Matej Hyčko	Auditor
Artem Delikatnyi	Auditor
Yulian Kisil	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

Marginfi's Solana contracts implement a margin lending and risk-managed borrowing stack. The system is centered around group-level configuration and per-asset banks that together support deposits, borrows, repayments, withdrawals, liquidations, flashloan flows, and integrations with external lending venues.

Marginfi Group is the protocol-level configuration and control layer. It stores global parameters and authorities, and governs how banks are created/configured (including oracle setup, limits, emissions, rate limits, and emergency controls).

Bank is the per-asset liquidity and accounting layer. Each bank maintains vaults, interest-rate state, risk parameters, and oracle configuration for a specific asset. User actions (deposit/borrow/repay/withdraw) update bank and account state according to risk checks and utilization-driven interest accrual.

Marginfi Account is the user-position layer. It tracks a user's balances across

banks and is the primary state used for health computation and solvency checks. Most user-facing actions route through this account abstraction and enforce collateral, liability, and liquidation constraints.

Risk & Health Engine is the solvency enforcement layer. It computes account health using configured risk weights and oracle prices, gates borrow/withdraw operations, and drives liquidation/deleverage paths when accounts breach requirements.

Oracle & Price Cache is the pricing input layer. Banks rely on configured oracle feeds and cached prices for risk checks, liquidation decisions, and accounting transitions. Correctness and freshness of these inputs directly affect solvency enforcement.

Liquidation / Deleverage Flow is the bad-debt mitigation layer. It coordinates start/end liquidation and deleverage transitions, repayment against unhealthy liabilities, and related state updates intended to restore account health while respecting configured constraints.

Operational Controls include pause and configuration mechanisms. Admin/authorized roles can configure group/bank parameters, update fee and staked settings, manage limits, collect/withdraw protocol fees and insurance, and trigger panic pause/unpause controls for incident response.

External Integrations extend bank behavior to other protocols. Marginfi includes integration paths for venues such as **Kamino**, **JupLend**, **Solend**, and **Drift**, plus staked-collateral settings/flows. These integrations introduce additional CPI dependencies and external-account assumptions into selected deposit/withdraw/position-management paths.

Trust Model

The audited scope relies on multiple privileged roles, oracle feeds, and external protocol integrations. Users must trust the following

parties/components not to abuse their privileges or publish incorrect state.

Marginfi Group

Users must trust that:

- the **group admin/config authorities** are correctly set and not compromised;
- global parameters (fees, limits, emergency controls, staked settings) are configured safely and not maliciously;
- bank onboarding/configuration decisions are valid for the intended asset/risk profile; and
- upgrade and governance processes do not introduce unauthorized behavior.

Banks (Per-Asset Markets)

Users must trust that:

- **bank configuration** (risk weights, limits, interest curve, emissions, oracle setup) is correct and consistent with policy;
- vault and accounting transitions enforce intended invariants for deposit/borrow/repay/withdraw flows;
- interest accrual and fee accounting are applied correctly and cannot be abused via parameter misconfiguration; and
- per-bank operational controls are used correctly and not for abusive censorship or fund extraction.

Marginfi Accounts / User State

Users must trust that:

- account initialization and ownership controls correctly bind authority to

the intended user;

- balance transitions and close/freeze mechanics cannot be abused to bypass safety checks; and
- cross-bank position state is updated atomically where required for risk correctness.

Risk & Liquidation Logic

Users must trust that:

- health computation and requirement checks (initial/maintenance/liquidation contexts) are implemented correctly;
- liquidation/deleverage paths enforce intended constraints and cannot be bypassed or griefed to create unfair losses;
- bankruptcy/bad-debt handling paths are properly authorized and operationally safe; and
- rate-limit and deleverage controls are configured to reduce systemic risk rather than introduce denial-of-service edge cases.

Oracle Inputs

Users must trust that:

- **oracle providers/reporters** publish accurate prices and do not maliciously manipulate updates;
- oracle freshness/confidence assumptions used by the protocol are appropriate for volatile markets; and
- oracle configuration per bank is correct so risk checks and liquidations use the intended feeds and precision rules.

Operational / Emergency Authorities

Users must trust that:

- **pause/unpause authorities** act in line with incident-response expectations and do not abuse controls;
- fee/insurance withdrawal authorities are correctly scoped and operated; and
- permissionless maintenance paths (where present) cannot be abused to violate core invariants.

External Integrations (Kamino, JupLend, Solend, Drift, Staked Flows)

Users must trust that:

- integration-specific accounts/configuration are set correctly and map to the intended external markets;
- external protocols enforce their own invariants and are not compromised or misconfigured;
- CPI account wiring and assumptions remain valid across dependency upgrades; and
- integration-specific failure modes (including availability and pricing dependencies) are accounted for in operational controls.

Fuzzing

Fuzz testing is an automated testing technique that discovers bugs by providing invalid, unexpected, or random data as inputs to a Solana program. Fuzz testing helps identify edge cases and vulnerabilities that might be missed during manual review.

Manually guided, stateful fuzz testing was utilized to test the correctness and robustness of the system. This approach involves guiding the fuzzer by specifying which Solana instruction sequences should be executed and what

accounts should be used, ensuring coverage of critical protocol paths rather than relying on purely random exploration.

For this audit revision, fork-based testing was utilized to create a testing environment that closely mimics Solana Mainnet behavior for key dependencies. Using Trident's fork feature, external accounts (notably Pyth Push oracle `PriceUpdateV2` accounts, and selected integration/program accounts) are fetched from mainnet and cached locally for reuse by the TridentSVM execution environment. This allows the fuzz tests to interact with realistic account layouts and dependencies without mocking.

Additionally, property-based fuzz testing was employed to verify the correctness and expected behavior of Solana program instructions. The fuzz harness enforces invariants after instruction execution, including:

- Conservation and directionality of SPL token transfers between user token accounts and bank vaults across deposit/withdraw/borrow/repay flows.
- Correct share-direction updates for assets and liabilities on successful operations.
- Flashloan closed-loop behavior (matching borrow/repay succeeds without net token drift; mismatches revert).
- Liquidation and receivership-session correctness under controlled oracle price shocks and time warps, including Pyth Push staleness handling by aligning `publish_time` with the fuzzed clock after time warps.
- Integration CPI flows (Kamino, JupLend) maintain expected token-leg conservation across user accounts, marginfi vaults, and integration reserve/vault accounts.
- Interest accrual behavior after time manipulation (bank `last_update` advances after a time warp + accrue).

The complete list of implemented execution flows and invariants is available in Appendix B.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

H1: Receivership liquidation bypass via dangling LiquidationRecord linkage

High severity issue

Impact:	Medium	Likelihood:	High
Target:	programs/marginfi/src/instructions/marginfi_account/liquidate_start.rs	Type:	Logic error

Description

The `marginfi_account_init_liq_record` logic writes `marginfi_account.liquidation_record` only once, while the corresponding `liquidation_record` Program Derived Address (PDA) cannot be closed or reinitialized.

If Alice closes a `marginfi_account` account and then reinitializes the same account address, the new account state starts with the default `liquidation_record` field value. However, the old `liquidation_record` PDA still exists for the same seed, so re-running the `InitLiquidationRecord` instruction fails because the PDA is already in use.

The receivership entry paths (`StartLiquidation` and `StartDeleverage`) enforce `has_one = liquidation_record` on the `MarginfiAccount` account, so they fail when this stale linkage exists. As a result, an account identity can become ineligible for receivership liquidation or deleverage, while classic liquidation via `lending_account_liquidate` remains available.

Listing 1. Excerpt from liquidate_start

```
217 has_one = liquidation_record @ MarginfiError::InvalidLiquidationRecord,
```

Exploit scenario

1. Alice, a user, initializes a `marginfi_account` account.
2. Alice calls the `InitLiquidationRecord` instruction.
3. A `liquidation_record` PDA is created and linked to `marginfi_account.liquidation_record`.
4. Alice closes her `marginfi_account` account.
5. Alice reinitializes the same `marginfi_account` account address.
6. The recreated account has the default `liquidation_record` field value.
7. Alice cannot call the `InitLiquidationRecord` instruction again because the old `liquidation_record` PDA still exists at the same seed and the `init` path fails.
8. Bob, a liquidator or risk admin, attempts the `StartLiquidation` instruction or the `StartDeleverage` instruction, but the calls fail due to the `has_one = liquidation_record` constraint mismatch.

Recommendation

Implement a close/reinitialize flow for the `liquidation_record` PDA that safely resets `marginfi_account.liquidation_record`.

Fix 1.1

The client fixed the issue by adding a `close_liquid_record` instruction in commit `2e5953f`[1] and by adding a check in the `marginfi_account` close path in commit `390dae2`[2]. The close path now verifies whether `marginfi_account.liquidation_record == Pubkey::default()`, preventing a `marginfi_account` account from being closed while it still references an open liquidation record and avoiding the stale default-address state after reinitialization.

Go back to Findings Summary

H2: `juplend_deposit` and `juplend_withdraw` do not invoke the rate limiter

High severity issue

Impact:	Medium	Likelihood:	High
Target:	programs/marginfi/src/instructions/juplend/deposit.rs, programs/marginfi/src/instructions/juplend/withdraw.rs	Type:	Logic error

Description

Neither JupLend deposit nor JupLend withdraw contains any `bank.rate_limiter. / group.rate_limiter.` calls, no `should_skip_rate_limit`, and no `apply_untracked_inflow`. Every other deposit/withdraw path (core, Kamino, Drift) participates in both bank- and group-level USD rate limits.

The admin-configured bank and group USD rate limits are therefore silently ignored for JupLend flows — a ready-made bypass of the protocol's circuit breaker during oracle or market incidents. Additionally, any `untracked_inflow` accumulated elsewhere is never applied against the group limiter through this path, skewing the aggregate accounting.

Exploit scenario

1. During an oracle incident, Alice (the admin) lowers group-wide outflow capacity to slow a brewing liquidation cascade.
2. Bob routes the liquidity he wants to pull through JupLend-backed banks via `juplend_withdraw`.
3. The rate limiter is never consulted; Bob's withdrawal proceeds at full size regardless of Alice's configured cap.

4. The protocol's circuit breaker is bypassed for the entire JupLend asset class.

Recommendation

Port the standard rate-limiter block from `marginfi_account/deposit.rs` and `marginfi_account/withdraw.rs`: `record_inflow` / `record_untracked_inflow` on deposit, and `try_record_outflow` + `apply_untracked_inflow` on withdraw. Denominate in underlying (`token_amount`) for consistency with `deposit_limit` and the oracle price.

Fix 1.1

The issue was fixed in commit `8f38cfb`[3]. Both `juplend_deposit` and `juplend_withdraw` now invoke the rate limiter through the `record_deposit_inflow` and `record_withdrawal_outflow` helpers.

Go back to Findings Summary

H3: JupLend-backed orders cannot be executed permissionlessly

High severity issue

Impact:	High	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/juplend/withdraw.rs	Type:	Logic error

Description

`start_execute_order` sets `ACCOUNT_IN_ORDER_EXECUTION`, and `validate_instructions` allowlists `JUPLEND_WITHDRAW` inside order transactions. But `JuplendWithdraw` uses the bespoke constraint

```
a.authority == authority.key() || a.get_flag(ACCOUNT_IN_RECEIVERSHIP)
```

which does not honor `ACCOUNT_IN_ORDER_EXECUTION`. Native, Kamino, and Drift withdraw paths use `is_signer_authorized(…, true, true)`, which does. Consequently a keeper-driven order execution of a JupLend asset leg always fails at the withdraw step.

Users can place (and pay order-init fees for) orders on JupLend-backed collateral, but when the trigger fires while they are offline — the exact scenario orders exist for — the stop-loss or take-profit does not execute. The protocol advertises automated loss-prevention for an asset class that silently opts out of it.

Exploit scenario

1. Alice places a stop-loss order on her JupLend-backed collateral and goes offline.

2. The market moves against her, and keeper Bob observes the trigger condition.
3. Bob calls `start_execute_order` followed by `juplend_withdraw`; the second call reverts with `Unauthorized` because Bob is neither `a.authority` nor does the account carry `ACCOUNT_IN_RECEIVERSHIP`.
4. The order cannot fire, and Alice's position stays exposed until she comes back online.

Recommendation

Replace the custom constraint with the canonical pair `account_not_frozen_for_authority` + `is_signer_authorized(…, true, true)`, matching the core, Kamino, and Drift withdraw paths. This fix overlaps with the freeze and rate-limit problems on the same instruction (M14, L20).

Fix 1.1

The issue was fixed in commit `9e8a881`[4]. The bespoke `JuplendWithdraw` constraint was replaced with the canonical pair `account_not_frozen_for_authority` plus `is_signer_authorized(.., true, true)` (matching the core, Kamino, and Drift withdraw paths), so a keeper running an order execution is now correctly authorized via the `ACCOUNT_IN_ORDER_EXECUTION` branch and JupLend-backed orders can execute permissionlessly.

Go back to Findings Summary

H4: Permissionless `settle_emissions` on `ACCOUNT_DISABLED` balances permanently drains `bank.emissions_remaining`

High severity issue

Impact:	Medium	Likelihood:	High
Target:	programs/marginfi/src/instructions/marginfi_account/emissions.rs, programs/marginfi/src/state/marginfi_account.rs, programs/marginfi/src/instructions/marginfi_group/handle_bankruptcy.rs	Type:	Access control

Description

`LendingAccountSettleEmissions` takes only `marginfi_account` (mut) and `bank` (mut), with a single `marginfi_account.group == bank.group` constraint. There is no signer, no `has_one = group`, no pause check, no session-flag gate, and no `ACCOUNT_DISABLED` / `ACCOUNT_FROZEN` check. The handler mutates `balance.emissions_outstanding`, `balance.last_update`, `bank.emissions_remaining`, and `marginfi_account.last_update`. As a result, any third party can advance emissions bookkeeping during a protocol pause, during a flashloan/order-execution/receivership session (mutating bank emissions state while the session's invariants are in flight), and, most damagingly, against `ACCOUNT_DISABLED` accounts.

`handle_bankruptcy` sets `ACCOUNT_DISABLED` on an account while leaving non-bankrupt balances open. Because `lending_account_settle_emissions` does not check `ACCOUNT_DISABLED` (unlike both `withdraw_emissions` variants, which do), a third party can call it against a disabled account's still-emissions-eligible

balance: `claim_emissions` decrements `bank.emissions_remaining` and increments `balance.emissions_outstanding`, but the tokens can never leave the balance because both withdraw paths reject disabled accounts.

The effect is a permanent, irreversible drain of the shared `bank.emissions_remaining` budget. Healthy depositors receive less over time because `claim_emissions` caps subsequent accruals by the depleted budget. A single transaction per disabled account suffices, and it is repeatable across all disabled accounts. See also M12.

Exploit scenario

1. Alice's account is disabled by bankruptcy handling but still has an open, emissions-eligible balance on bank `B`.
2. Bob calls `lending_account_settle_emissions` against Alice's balance.
3. `claim_emissions` debits `B.emissions_remaining` and credits `Alice.balance.emissions_outstanding`.
4. Neither withdraw path releases those tokens (both reject `ACCOUNT_DISABLED`); they are permanently locked in the vault.
5. The budget available to healthy depositors is permanently reduced.

Recommendation

Add `group: AccountLoader<'info, MarginfiGroup>` to `LendingAccountSettleEmissions` with `has_one` on both `marginfi_account` and `bank`, add the pause constraint, and add `check!(!marginfi_account.get_flag(ACCOUNT_DISABLED), MarginfiError::AccountDisabled)` (and optionally the session flags). Separately, add a sweep path that refunds `emissions_outstanding` of disabled accounts back to `bank.emissions_remaining`.

Fix 1.1

The issue was fixed in commit `f28865d`[5]. `lending_account_settle_emissions` was removed entirely; the only remaining permissionless emissions instruction is `lending_account_clear_emissions`, which can be called only after a bank's emissions have been disabled (`emissions_rate == 0` and `EMISSION_FLAGS == 0`) and which simply zeroes the per-balance `emissions_outstanding` tracking field — no transfers and no debit of `bank.emissions_remaining` occur. The client added the following comment.

> *emissions are wind-down, functions mentioned in descriptions are no longer in codebase*
>
> — Project0 team

Go back to Findings Summary

M1: Drift withdraw USD valuation double-counts deposit interest

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/drift/withdraw.rs	Type:	Logic error

Description

When group-level rate limiting is enabled, `drift_withdraw` computes a USD outflow value using `calc_value(token_amount, price, bank.mint_decimals, None)`. For Drift banks, both of these inputs already incorporate Drift's `cumulative_deposit_interest`, so the USD value is inflated by approximately the interest factor.

- The Drift oracle adapter (`OracleSetup::DriftPythPull`) scales the oracle price by `spot_market.cumulative_deposit_interest` (i.e., the oracle price reflects the Drift deposit exchange rate rather than the underlying asset's "base" price).
- The withdrawal token amount is computed via `MinimalSpotMarket::get_withdraw_token_amount(scaled_balance)`, which also multiplies by `cumulative_deposit_interest` to convert Drift "scaled balance" into native tokens.

As a result, when `drift_withdraw` records the group USD outflow, it effectively applies the deposit interest twice.

Example (USDC): A user deposits 1,000 USDC into Drift and later accrues 3% deposit interest. On withdraw-all, `token_amount = 1,030`. Because the Drift oracle adapter also scales the price, `price = 1.03`. The group rate limiter is charged approximately \(1030 * 1.03 = 1060.9\) USD instead of the correct

\(1030\) USD.

This does not change the actual number of tokens transferred to the user, but it can incorrectly reject withdrawals (and any other flows that rely on the group USD limiter) by making the limiter believe more USD left the system than actually did.

Exploit scenario

1. Alice, a governance or admin actor, enables group-level USD rate limiting.
2. Bob deposits into a Drift bank and time passes so `cumulative_deposit_interest > 1`.
3. Bob withdraws (or a keeper withdraws during order execution / receivership flows that require oracle pricing).
4. The instruction computes `usd_value` using an interest-adjusted `token_amount` and an interest-adjusted `price`, inflating the recorded USD outflow.
5. The group limiter can be exceeded prematurely, causing Bob's withdrawal to revert even though the true USD outflow is within limits.

Recommendation

Ensure Drift's deposit exchange-rate is applied exactly once when computing USD value for group rate limiting.

In `drift_withdraw`, when computing the group USD outflow: - Prefer valuing the **scaled** amount (e.g., `expected_scaled_balance_change`) against the **interest-adjusted** Drift oracle price, using `bank.get_balance_decimals()` (Drift scaled-balance decimals), OR - Use the **token** amount (`token_amount`) against the **unadjusted** base oracle price (i.e., do not apply `cumulative_deposit_interest` inside the oracle adapter for this valuation path).

Add a regression test that sets `cumulative_deposit_interest > 1`, withdraws

from a Drift bank with the group limiter enabled, and asserts the recorded USD outflow equals the true token outflow (not multiplied by the interest factor twice).

Fix 1.1

The issue was fixed in the commit `ef77688`[6] by updating `record_withdrawal_outflow` to value group USD outflow using the scaled balance decrement (`expected_scaled_balance_change`) and `bank.get_balance_decimals()` instead of the native token amount and mint decimals, so the interest-adjusted Drift oracle price is applied exactly once.

Go back to Findings Summary

M2: Permissionless `migrate_curve` can retroactively re-rate unaccrued interest

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/marginfi_group/migrate_curve.rs	Type:	Logic error

Description

`migrate_curve` is permissionless and mutates `bank.config.interest_rate_config` in-place, but it does not call `accrue_interest` before changing the curve representation. Since migration quantizes legacy parameters into new bucketed fields, the post-migration curve is not guaranteed to be bit-equivalent to the pre-migration curve.

As a result, the next time interest accrues for the interval \([bank.last_update, now]\), the accrued interest for that entire interval is computed using the new curve, not the curve that was active during the interval. A user (especially a borrower) can time the permissionless migration to slightly reduce the interest they pay (and correspondingly reduce depositor yield) for the unaccrued period.

Exploit scenario

1. A bank has not accrued interest recently (non-zero ((now - last_update)\)).
2. Alice, a borrower, observes that migration-induced quantization would slightly reduce the effective rate.
3. Alice calls `migrate_curve` permissionlessly immediately before repaying or closing.

4. The subsequent accrual uses the post-migration curve for the entire pending interval, reducing Alice's interest owed and reducing Bob's depositor earnings compared to accruing under the pre-migration curve.

Recommendation

- Accrue interest before mutating the curve config. This likely requires passing `MarginfiGroup` (and any required accounts) into `MigrateCurve` so `bank.accrue_interest(…)` can run.
- Add the standard protocol pause / bank state gating (`!group.is_protocol_paused()`, `validate_bank_state(…)`) to avoid permissionless migration during paused/restricted states.

Fix 1.1

The issue was fixed in the commit `4a20de1`[7] by removing the permissionless `migrate_curve` instruction entirely and deprecating the legacy curve type, so users can no longer retroactively re-rate unaccrued interest through a permissionless migration. Interest accrual now rejects legacy curve banks, and new banks are created with the seven-point curve format from the start.

Go back to Findings Summary

M3: Bank creation allows unbounded Token-2022 mint extensions

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/add_pool.rs	Type:	Configuration

Description

Bank creation paths accept `bank_mint` and initialize vaults against it, but they do not enforce an allowlist of supported extension sets during bank onboarding.

In practice, later execution paths handle selected extension behavior (for example transfer-fee logic), but onboarding does not reject mints that include other Token-2022 extension combinations. This leaves compatibility and safety guarantees dependent on admin choices rather than explicit protocol-level constraints.

Exploit scenario

1. Alice, a protocol admin, creates a new bank using a Token-2022 mint that includes extension combinations not explicitly constrained at onboarding.
2. The `lending_pool_add_bank` instruction succeeds because onboarding does not enforce an extension allowlist.
3. Bob, a user, interacts with the new bank under assumptions that listed assets follow expected mint behavior.
4. A downstream path encounters extension behavior outside those assumptions, creating operational risk or unexpected protocol behavior.

Recommendation

Enforce a mint allowlist policy at bank creation, including allowed extension types.

Acknowledgment 1.1

The client acknowledged the finding and provided the following comment:

> *In our risk model, the admin decides which tokens can be used in a bank, and this is validated by our internal process.*
>
> — Project0 team

Go back to Findings Summary

M4: Permissionless propagation can override manual staked bank configuration

Medium severity issue

Impact:	Low	Likelihood:	High
Target:	programs/marginfi/src/instructions/marginfi_group/propagate_staked_settings.rs	Type:	Access control

Description

The `propagate_staked_settings` instruction is permissionless and rewrites multiple `bank.config` fields for any staked bank in the group.

As a result, manual parameter changes made through admin configuration can be overwritten by any caller at any time by re-applying group-level `staked_settings` to the bank.

The main mitigation is that only authorized actors can modify the `staked_settings` account itself.

Listing 2. Excerpt from propagate_staked_settings

```
36 pub struct PropagateStakedSettings<'info> {
37     pub marginfi_group: AccountLoader<'info, MarginfiGroup>,
38
39     #[account(
40         has_one = marginfi_group @ MarginfiError::InvalidGroup
41     )]
42     pub staked_settings: AccountLoader<'info, StakedSettings>,
43
44     #[account(
45         mut,
46         constraint = {
47             let bank = bank.load()?;
48             bank.group == marginfi_group.key() &&
49                 bank.config.asset_tag == ASSET_TAG_STAKED
```

```
50         }
51     )]
52     pub bank: AccountLoader<'info, Bank>,
53 }
```

Exploit scenario

1. Alice, a protocol admin, manually updates a staked bank parameter through an admin-only bank configuration path.
2. Bob, any external user, calls the `propagate_staked_settings` instruction for the same bank.
3. The instruction permissionlessly rewrites the bank fields from `staked_settings`.
4. Alice's manual bank-level change is reverted, and Bob can repeat this action whenever manual changes are applied again.

Recommendation

Restrict propagation to an authorized role or add a per-bank opt-out for manual configuration.

Acknowledgment 1.1

The client acknowledged the finding.

Go back to Findings Summary

M5: `juplend_withdraw` does not validate remaining-account ordering on `withdraw_all`

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/juplend/withdraw.rs	Type:	Data validation

Description

Kamino, Drift, and the core withdraw instruction all validate remaining-account ordering when `withdraw_all` is true, so that the closing balance's risk accounts appear last — preventing `check_account_init_health` from pairing the wrong bank's risk accounts to the wrong balance after the post-close sort. JupLend omits this guard.

Under `withdraw_all`, the caller can choose an adversarial ordering of `remaining_accounts`, causing the post-sort health engine to mis-associate risk accounts with balances. Depending on the specific sort outcome, this can either allow a withdraw that should fail health to succeed, or cause a legitimate withdraw to revert.

Exploit scenario

1. Alice calls `juplend_withdraw` with `withdraw_all = true`. Bob expects `withdraw_all` to preserve the same remaining-account ordering guarantees enforced by Kamino, Drift, and the core withdraw instruction.
2. Alice crafts a `remaining_accounts` order that places her closing balance's risk accounts out of position.
3. After the post-close sort, the health engine evaluates the closing-balance risk accounts against an unrelated balance's shares.

4. Alice's withdraw passes a health check it should have failed, letting her take out more than her true post-withdraw health allows; depending on the specific sort outcome, Bob's legitimate withdraw can also revert.

Recommendation

At the top of `juplend_withdraw`, when `withdraw_all` is true, invoke the standard `validate_remaining_accounts_for_balances_close_last` helper against the account's `lending_account`, the context's `remaining_accounts`, and the current `bank_key`, matching the guard already used by Kamino, Drift, and the core withdraw instruction.

Fix 1.1

The issue was fixed in commit `8f38cfb`[8]. Rather than add the missing `validate_remaining_accounts_for_balances_close_last` call to JupLend, the client removed the equivalent risk-account-ordering enforcement from the `withdraw_all` / `repay_all` paths across all integrations (core, Kamino, Drift, and JupLend), making the behavior uniform across all withdraw paths. The client added the following comment.

> *remove account risk accounts enforcement for withdraw_all / repay_all paths*
>
> — Project0 team

Go back to Findings Summary

M6: Stop-loss slippage protection degrades when position is underwater

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/marginfi_account/order.rs	Type:	Logic error

Description

When a keeper triggers a stop-loss order, `start_execute_order` overwrites `order.stop_loss` with the current `net` value (asset equity minus liability equity) at execution time. The user's original stop-loss threshold is discarded.

In `end_execute_order`, the slippage check computes:

```
let allowed_sl = sl.checked_mul(slippage_frac)?;
check!(net >= allowed_sl, MarginfiError::OrderExecutionOverWithdrawal);
```

where `sl` is the already-overwritten value (the net at trigger time) and `slippage_frac = 1 - slippage_bps / 10_000`.

Because the slippage fraction is applied to the degraded net rather than the original threshold, the slippage parameter becomes meaningless once the position has moved significantly below the threshold. When `net` is zero or negative, `allowed_sl` becomes zero or negative, and the check is trivially satisfied regardless of how much the keeper extracts.

The same `order.stop_loss = net.into()` assignment also runs on the `OrderTriggerType::Both` arm when the take-profit leg triggered. There is no direct exploit today because the order is closed at `end_execute_order` and nothing else reads `order.stop_loss` between `start` and `end`, but the written

value is semantically wrong on the TP branch and any future refactor that introduces a retry path, recovery flow, or any read of `order.stop_loss` between `start` and `end` will silently misbehave. The fix below (stop overwriting `order.stop_loss` in `start_execute_order`) addresses both the exploit path and this latent case.

Exploit scenario

1. Alice places a stop-loss with threshold `$50` and slippage `5%`.
2. Alice's position degrades so that `net = $2` at execution time.
3. Keeper Bob calls `start_execute_order`, which overwrites `order.stop_loss = $2`.
4. The slippage check in `end_execute_order` becomes `net >= $2 * 0.95 = $1.90`.
5. Bob extracts nearly the entire remaining value; Alice receives `$1.90` instead of the ~5% slippage around `$50` she expected.

Recommendation

Preserve the user's original `stop_loss` threshold through order execution and apply the slippage fraction to that preserved value rather than to the degraded `net` at trigger time. Concretely, stop overwriting `order.stop_loss` in `start_execute_order`, and in `end_execute_order` compute the allowed floor as the original threshold multiplied by `slippage_frac` and enforce `net >= allowed_sl`, raising `MarginfiError::OrderExecutionOverWithdrawal` otherwise. Alternatively, document clearly that the slippage parameter protects only against execution slippage relative to the position's value at trigger time, not against position degradation between order creation and execution.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *this is expected behaviour, cranker working as expected*
>
> — Project0 team

Go back to Findings Summary

M7: Rate-limit inflow uses cached price while outflow uses live oracle, enabling capacity rental

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/utils/general.rs, programs/marginfi/src/state/rate_limiter.rs, programs/marginfi/src/instructions/marginfi_account/{deposit,repay,withdraw,borrow}.rs	Type:	Logic error

Description

Deposits and repays fetch the USD conversion price for the rate limiter from the bank's cache (`fetch_rate_limit_price_for_inflow`); withdraws and borrows use the live oracle. If the cache is invalid, deposits buffer the inflow as the native amount via `record_untracked_inflow`, and the next outflow converts that buffered amount at the live oracle price — i.e. inflow is USD-valued at a moment the attacker can time.

Combined with the USD-truncation and signed-window-counter issues in the limiter, this enables cheap rental of rate-limit capacity: an attacker deposits when the cached price is stale or low, then withdraws when the live oracle price has moved, receiving more USD-denominated capacity than the inflow should have granted.

Exploit scenario

1. Alice observes that bank `B's cached rate-limit price is stale and below the

live oracle.

2. Alice deposits `X` units; because the cache is invalid, the inflow is buffered as the native amount.
3. Moments later, `apply_untracked_inflow` converts `X` at the live (higher) oracle price and credits it against the group's rate-limit window.
4. Alice withdraws more USD-denominated capacity than her deposit actually funded, crowding out Bob's legitimate outflows under the same group limiter window.

Recommendation

Store USD value (not native amount) at inflow time using the cached price. If no valid cached price is available, reject the inflow instead of buffering it for later conversion at a different price.

Fix 1.1

The issue was fixed in commit `8f38cfb`[9]. The rate limiter was refactored so that group-level (USD) accounting is now handled off-chain by a delegated admin via `update_group_rate_limiter`; on-chain code only emits `RateLimitFlowEvent` and tracks the bank limiter in native units, and the buggy `apply_untracked_inflow` path no longer exists. The cached-vs-live USD price disparity is therefore no longer possible. The client added the following comment.

> *With rate limit refactored this is no longer an issue*
>
> — Project0 team

Go back to Findings Summary

M8: `lending_pool_accrue_bank_interest` is not gated by the protocol pause

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/marginfi_group/accrue_bank_interest.rs	Type:	Access control

Description

All other ops instructions (`collect_bank_fees`, `withdraw_fees*`, `withdraw_insurance`, `update_fees_destination_account`) enforce `is_protocol_paused`, but `accrue_bank_interest` does not. A permissionless cranker can therefore grow liabilities against borrowers during the pause window — while those borrowers cannot repay, because `repay` is pause-gated. The pause becomes asymmetrically punitive: debt continues to grow but cannot be reduced.

Exploit scenario

1. Alice, the admin, activates `panic_pause` during a market incident.
2. Bob, a keeper or adversary, continuously calls `accrue_bank_interest` against a high-utilization bank while the pause is active.
3. Borrowers cannot call `repay` because `repay` is pause-gated.
4. The pause lifts; borrowers owe strictly more than when the pause started, despite having no avenue to reduce their debt during the window.

Recommendation

Add `constraint = !group.load()?.is_protocol_paused() @ MarginfiError::ProtocolPaused` to the `group` account, consistent with the

other ops instructions. Alternatively, exclude paused time from `time_delta` inside `accrue_interest`.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *We probably like to keep it as it is, so even on-paused interest is accumulated*
>
> — Project0 team

Go back to Findings Summary

M9: `keeper_close_order` / `close_order` rent refund goes to caller-chosen `fee_recipient`

Medium severity issue

Impact:	Low	Likelihood:	High
Target:	programs/marginfi/src/instructions/marginfi_account/order.rs	Type:	Access control

Description

The `Order` rent was paid by the original `fee_payer` at `place_order` time. `KeeperCloseOrder` and `CloseOrder` both accept an unchecked `fee_recipient` and close the PDA with `close = fee_recipient`. Whenever the order becomes keeper-closeable (or the `marginfi_account` is already closed), any MEV searcher can sweep the rent to themselves instead of returning it to the original funder.

Worse, when `marginfi_account` has already been closed, `keeper_close_order` takes a special "can close = true" path with `authority_pk = Pubkey::default()` and `group_pk = Pubkey::default()` — the emitted event then carries bogus fields (see I14).

Exploit scenario

1. Alice pays the rent to place an order that later becomes keeper-closeable.
2. Bob, a searcher, calls `keeper_close_order` with `fee_recipient = bob_wallet`.
3. The Anchor `close = fee_recipient` drains the rent into Bob's wallet.
4. Alice never recovers the rent she originally paid.

Recommendation

Store the original `fee_payer` on `Order` at `place_order` time and pin `close = <stored_fee_payer>`, or target `FeeState.global_fee_wallet`. Either way, remove caller discretion over the rent recipient.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *this is false positive, rent is incentive for keepers to close orders*
>
> — Project0 team

Go back to Findings Summary

M10: Close-path permanently strands outstanding emissions (both whole units and sub-unit dust)

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/marginfi_account/close.rs, programs/marginfi/src/state/marginfi_account.rs, type-crate/src/types/user_account.rs	Type:	Logic error

Description

`emissions_outstanding` is debited from `bank.emissions_remaining` at accrual time. Both the account-level close path and the balance-level close path destroy the bookkeeping without paying the user or refunding the bank, so outstanding emissions are permanently stranded. The stranding happens at two different magnitudes, driven by the same structural gap.

Whole-unit strand on `close_account`. `can_be_closed()` uses `balance.get_side().is_none()` as its "empty" test. `get_side()` returns `None` when both `asset_shares` and `liability_shares` are below `EMPTY_BALANCE_THRESHOLD`, but it does not consult `emissions_outstanding`. `Balance::close(true)` refuses to close a balance with `emissions_outstanding >= 1` — that invariant is bypassed at the account-close path. A user who does an exact `withdraw` / `repay` that zeroes shares while leaving the balance `active = true` can immediately call `close_account`: `can_be_closed()` returns true, Anchor's `close = fee_payer` destroys the marginfi account, and the

outstanding emissions (already debited from `bank.emissions_remaining`) are lost — tokens stay locked in the emissions vault with the counter at zero, so nobody can claim them.

Sub-unit dust strand on `balance.close(true)`. `balance.close(true)` permits close when `emissions_outstanding < 1`, then resets the balance to `empty_deactivated()`, zeroing the `emissions_outstanding` field. Any fractional emission is discarded — it has been debited from `bank.emissions_remaining` but is never paid to the user and never returned to the shared budget.

The two cases share the same root cause — close paths destroy `emissions_outstanding` without either paying it out or refunding it — and share the same fix shape.

Exploit scenario

1. Alice accrues `5` units of `emissions_outstanding` on a bank.
2. Alice does a clean `withdraw_all` that zeroes her shares.
3. Alice immediately calls `close_account`; `can_be_closed()` returns true because `get_side()` sees both sides below threshold.
4. The account is destroyed and the `5` outstanding units are permanently stranded in the emissions vault; `bank.emissions_remaining` has already been debited, so Bob and other users cannot claim them either.

The sub-unit case follows the same shape on `balance.close(true)`: the fractional portion (`< 1`) is silently zeroed at close and becomes unreachable.

Recommendation

Couple close to emissions settlement on both paths. Extend `can_be_closed()` (or add a pre-check in `close_account`) to require `balance.emissions_outstanding < 1` for every balance — mirroring `Balance::close(true)` — and in `Balance::close(true)` itself, refund any

fractional `emissions_outstanding` to `bank.emissions_remaining` (or settle it via `settle_emissions_and_get_transfer_amount`) before resetting the field. Alternatively, unconditionally settle and refund outstanding emissions on any close path before destroying state.

Fix 1.1

The issue was fixed in commit `f28865d`[10]. `Balance::close` no longer touches emissions and `claim_emissions` has been removed from `close_balance` and the account-level close path, so there is nothing to strand at close time. Outstanding emissions are now wound down out-of-band via `lending_pool_reclaim_emissions_vault` (admin reclaims vault tokens) and `lending_account_clear_emissions` (per-balance). The client added the following comment.

> *emissions are wind-down*
>
> — Project0 team

Go back to Findings Summary

M11: Operation-type enforcement checks use tolerance without clamping, allowing sub-threshold invariant violations

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/state/marginfi_account.rs, type-crate/src/constants.rs	Type:	Logic error

Description

`BalanceDecreaseType::WithdrawOnly` / `BorrowOnly` and `BalanceIncreaseType::RepayOnly` / `DepositOnly` each call `check!(x.is_zero_with_tolerance(ZERO_AMOUNT_THRESHOLD))` against the side they mean to prohibit, then proceed to book the original (non-zeroed) value into shares. Any value in the open interval `(0, 0.0001)` passes the guard and is written to state as real shares.

Worst-case paths combine with `ReduceOnly` semantics. `withdraw` and `repay` are allowed in `ReduceOnly` banks (`InstructionKind::FailsInPausedState`), so a sub-threshold `liability_amount_increase` during `withdraw` — or `asset_amount_increase` during `repay` — creates new liabilities or assets in a bank that is supposed to be reducible only. The operation is repeatable in small increments.

Consequences:

- `WithdrawOnly` invariant broken — `withdraw` can create liabilities.
- `RepayOnly` invariant broken — `repay` can create assets.
- `ReduceOnly` policy bypass via both paths, repeatable.

- `BorrowOnly` and `DepositOnly` also violated (consumes dust of the wrong side).

Per-call magnitude is bounded by `ZERO_AMOUNT_THRESHOLD` (0.0001 tokens), but the policy break is real.

Exploit scenario

1. Alice targets a `ReduceOnly` bank. Bob expects `ReduceOnly` banks to prohibit new liabilities or assets.
2. Alice calls `withdraw` with a carefully chosen amount such that the prohibited liability-increase side evaluates to a value in `(0, 0.0001)`.
3. The tolerance check passes, but the original non-zero value is written into `liability_shares`, creating a new liability in a reduce-only bank.
4. Alice repeats in small increments, accumulating a real liability position.

Recommendation

After the `is_zero_with_tolerance` check passes, clamp the value to `I80F48::ZERO` before computing shares and writing state, so tolerated values are treated as zero consistently. Apply to all four branches.

Fix 1.1

The issue was fixed in commit `186b248`[11]. After each `is_zero_with_tolerance(ZERO_AMOUNT_THRESHOLD)` guard in the `WithdrawOnly` / `BorrowOnly` / `RepayOnly` / `DepositOnly` branches, the tolerated "dust" value is now explicitly clamped to `I80F48::ZERO` before share conversion and state updates, so sub-threshold artifacts cannot be booked onto the prohibited side (including the repeatable `ReduceOnly` bypass pattern).

Go back to Findings Summary

M12: `emissions_destination_account` survives authority transfer — prior owner can drain post-transfer emissions

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/marginfi_account/transfer_account.rs, programs/marginfi/src/instructions/marginfi_account/emissions.rs, programs/marginfi/src/state/marginfi_account.rs	Type:	Access control

Description

`MarginfiAccount::initialize` sets `emissions_destination_account = Pubkey::default()`, but `transfer_to_new_account` and `transfer_to_new_account_pda` immediately overwrite the new account with `old_account.emissions_destination_account`. `lending_account_withdraw_emissions_permissionless` requires no signer from the current authority — it derives the ATA from the stored destination wallet and transfers.

Any post-transfer emissions can therefore be claimed to the prior owner's wallet by anyone (including the prior owner themselves) until the new authority notices and updates the field. The new owner may reasonably assume reward flow belongs to them after receiving authority; this assumption is silently violated.

Exploit scenario

1. Alice transfers her marginfi account to Bob via `transfer_to_new_account`.
2. The transfer copies Alice's `emissions_destination_account` onto Bob's new account.
3. Emissions continue to accrue on the transferred balances.
4. Alice (or any third-party keeper) calls `lending_account_withdraw_emissions_permissionless`; the tokens route to Alice's original wallet via the stored destination.
5. Bob eventually notices the misconfiguration and calls `marginfi_account_update_emissions_destination_account`, but by then Alice has skimmed everything accrued in the interim.

Recommendation

In both transfer paths, explicitly reset `new_account.emissions_destination_account = Pubkey::default()` instead of copying the old value. The new owner must then call `marginfi_account_update_emissions_destination_account` before `permissionless_withdraw_emissions` can route funds.

Fix 1.1

The issue was fixed in commit `f28865d`[12]. The drain vector is gone because every on-chain instruction that routed tokens to `emissions_destination_account` (the authority-only `lending_account_withdraw_emissions` and its permissionless variant) was removed; the field is now used only for off-chain accounting. Note that `transfer_to_new_account` and `transfer_to_new_account_pda` still literally copy `emissions_destination_account` from the old to the new account, but with the routing instructions gone this is no longer reachable as an exploit. The client added the following comment.

> *emissions are wind-down, functions mentioned are no longer in codebase*
>
> — Project0 team

Go back to Findings Summary

M13: Order execution snapshots stale pre-refresh health, enabling value extraction

Medium severity issue

Impact:	Low	Likelihood:	High
Target:	programs/marginfi/src/instructions/marginfi_account/order.rs, programs/marginfi/src/instructions/marginfi_account/liquidate_start.rs	Type:	Logic error

Description

`start_execute_order` snapshots `order_start_health` from whatever bank and reserve state happens to be active at that instruction. The in-transaction allowlist does not require oracle or rate refreshes to precede `start_execute_order`: Kamino, Drift, and JupLend refresh CPIs, along with `lending_account_withdraw` / `repay` (which call `accrue_interest`), can land between `start` and `end`. `end_execute_order` then evaluates the TP max-fee and the "health not worse" checks against the stale snapshot, allowing the delta between pre- and post-refresh value to be extracted as execution fee by the keeper.

Exploit scenario

1. Keeper Bob sequences an order-execution transaction as `start_execute_order` → `accrue_interest` / integration-refresh CPIs → `end_execute_order`.
2. The snapshot taken at `start` reflects stale pre-refresh values.
3. The mid-transaction refreshes move Alice's health upward (interest

accrued, Kamino / Drift / JupLend rate updates lift share values).

4. `end_execute_order` compares against the stale snapshot and charges the TP max-fee cap against an under-stated baseline.
5. Bob extracts the refreshed-minus-stale delta as execution fee.

Recommendation

Require refreshes (bank accrual, Kamino/Drift/JupLend rate CPIs) to appear before `start_execute_order` in the validator allowlist (an order-sensitive check in `validate_ix_first`), or snapshot after calling refresh helpers inline. Alternatively, re-compute `order_start_health` post-refresh inside `end_execute_order` and use the smaller of the two values.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *this is design tradeoff and CU save, liquidations*
>
> — Project0 team

Go back to Findings Summary

M14: `juplend_deposit` and `juplend_withdraw` bypass `ACCOUNT_FROZEN` and the canonical authorization pair

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/juplend/deposit.rs, programs/marginfi/src/instructions/juplend/withdraw.rs	Type:	Access control

Description

Every deposit and withdraw instruction in the protocol except the two JupLend wrappers guards `marginfi_account` with the canonical pair `account_not_frozen_for_authority` + `is_signer_authorized`. That pair correctly blocks a frozen authority, admits the group `admin` when the account is frozen, and (on the withdraw side) admits the receivership and order-execution flows. Both JupLend entry points substitute their own constraint and silently lose those properties, so a frozen user can still move JupLend-backed collateral and the admin has no remediation path.

`juplend_withdraw` — drain primitive. The `JuplendWithdraw` accounts struct uses a bespoke predicate `a.authority == authority.key() || a.get_flag(ACCOUNT_IN_RECEIVERSHIP)`. It omits any check for `ACCOUNT_FROZEN`: a user whose account has been frozen for remediation, seizure, or compliance remains the stored `authority` and still satisfies the constraint, so they can withdraw JupLend-backed collateral straight out from under the freeze. It also provides no admin path and no order-execution path — the group admin cannot act on a frozen account, and order-execution flows that set `ACCOUNT_IN_ORDER_EXECUTION` cannot withdraw either. This is an active

exfiltration path, not just a policy drift.

`juplend_deposit` — freeze bypass without drain. `JuplendDeposit` guards `marginfi_account` only with `has_one = authority`. The handler checks `ACCOUNT_DISABLED` and `ACCOUNT_IN_RECEIVERSHIP` but never consults `ACCOUNT_FROZEN`, and there is no `group.admin` pathway. A frozen user can still call `juplend_deposit`, and — symmetrically to the withdraw gap — the admin cannot deposit as part of a remediation flow.

Both instructions are variants of the same bug (custom constraint instead of the canonical pair) with the same fix shape. The withdraw variant is the more severe of the two because it enables exfiltration; the deposit variant undermines the freeze semantics in the other direction.

Exploit scenario

Withdraw-side exfiltration (the drain scenario):

1. Bob, the admin, freezes Alice's `marginfi_account`.
2. Alice calls `juplend_withdraw`; the custom constraint accepts her signature because `a.authority == authority.key()` remains true.
3. Alice drains her JupLend-backed collateral before Bob's remediation completes.

Deposit-side bypass (the policy scenario):

1. Bob, the admin, freezes Alice's `marginfi_account` for remediation.
2. Alice calls `juplend_deposit`; `has_one = authority` resolves and the freeze does not gate this path.
3. Bob attempts to move Alice's JupLend-backed funds via `juplend_deposit` as part of remediation, but no admin pathway exists and his call fails.

Recommendation

Replace both custom constraints with the standard pair used by every other deposit and withdraw instruction: an `account_not_frozen_for_authority` check against the loaded `marginfi_account` and the `authority` key that raises `MarginfiError::AccountFrozen`, followed by an `is_signer_authorized` check against the loaded `marginfi_account`, the group `admin`, and the `authority` key that raises `MarginfiError::Unauthorized`. On `juplend_deposit` pass both flags on `is_signer_authorized` as `false`, matching the stricter deposit-path policy used by `LendingAccountDeposit` and `KaminoDeposit`. On `juplend_withdraw` pass both flags as `true`, which preserves the receivership exemption already granted by the original predicate and additionally admits order-execution flows, matching the surrounding protocol design.

Fix 1.1

The issue was fixed in commit `9e8a881`[13]. Both `JuplendDeposit` and `JuplendWithdraw` now use `account_not_frozen_for_authority` and `is_signer_authorized` in their account constraints, matching the core / Kamino / Drift withdraw paths, so a frozen account's authority is now correctly blocked from both deposit and withdraw on the JupLend integration. The client added the following comment.

> *Fixed before report*
>
> — Project0 team

Go back to Findings Summary

M15: Stale orders can execute against unrelated positions because `Order` binds to reusable tags rather than stable bank identity

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	programs/marginfi/src/state/order.rs, programs/marginfi/src/state/marginfi_account.rs, programs/marginfi/src/instructions/marginfi_account/{close,initialize,order}.rs	Type:	Logic error

Description

An `Order` stores only `(marginfi_account_pubkey, tags[u16; ORDER_ACTIVE_TAGS], trigger, slippage)`. It does not store the original bank pubkeys, the group, or any account-incarnation/generation marker. Tags are reusable identifiers assigned by a `u16` allocator, so the `(marginfi_account, tag)` pair is not a stable binding: the same pair can resolve to different underlying balances over time. Two variants of this reuse lead to stale orders executing against positions the user never meant to protect.

Account reinit (cross-incarnation tag reuse). `marginfi_account_close` allows the account to be closed while its order PDAs remain open. For non-PDA (keypair-based) accounts the same address can later be reinitialized; the fresh account starts from zeroed state, so early tag assignments (`1`, `2`, …) are reused. The stale order's `has_one = marginfi_account` still resolves, and `start_execute_order` binds the order legs to whatever current balances carry those tags — attaching a pre-existing order to entirely different positions

under stale trigger/slippage. Because `Order` does not bind to group identity either, a non-PDA account address can even be recreated under a different group and still satisfy the stale order's constraints. This is the dominant exploit path.

Tag-space wraparound (same-incarnation tag reuse). `set_keeper_close_flags` can zero a balance's `tag` without closing the associated `Order` PDA — the `Order` then becomes stale but stays live. `reserve_n_tags` only excludes tags currently present on active balances; it does not exclude tags still referenced by open stale orders. `last_tag_used` is a `u16`, so after ~65k tag allocations on the same account the allocator wraps and hands the stale order's tags to unrelated current balances. `start_execute_order` then binds the stale order to the new balances under stale trigger/slippage. This variant is much narrower — it requires the user to actively churn orders past `u16` wraparound on the same account — but it rests on the same structural gap as the account-reinit variant.

Both variants resolve to the same root cause: `Order` identifies its legs by reusable tags on a mutable account, not by stable bank identity.

Exploit scenario

Account-reinit variant:

1. Alice places a stop-loss order on positions `(bankA, bankB)` in her non-PDA marginfi account; the tag allocator assigns tags `1` and `2`.
2. Alice closes the marginfi account but leaves the order PDA open.
3. Alice reinitializes a fresh marginfi account at the same keypair address and opens new positions in banks `(bankX, bankY)`, which receive tags `1` and `2` from the zeroed tag allocator.
4. Bob, a keeper, calls `start_execute_order` on the stale order; it resolves against the new `(bankX, bankY)` positions under the old threshold and

slippage.

5. The order triggers a loss-inducing exit on positions Alice never meant to protect.

Tag-wraparound variant:

1. Alice uses `set_keeper_close_flags` to zero tags on a balance whose associated `Order` PDA remains open.
2. Alice churns orders on her marginfi account long enough to consume the full `u16` tag space (~65k allocations); `last_tag_used` wraps.
3. `reserve_n_tags` hands the stale order's tag values to new balances because open stale orders are not in its exclusion set.
4. Bob, a keeper, calls `start_execute_order` on the stale order; it binds to Alice's current positions under the old trigger and slippage.

Recommendation

Fix the identity binding rather than patching each reuse path. Store a stable identity on `Order` — the original bank pubkeys, the original group, and/or an account-generation counter bumped on each `MarginfiAccount::initialize` — and verify it at `start_execute_order`. This resolves both the cross-incarnation reinit case and the same-account wraparound case. Defence-in-depth options that also help: require all `Order` PDAs associated with a `marginfi_account` to be closed before `marginfi_account_close` succeeds (eliminates the reinit variant), include tags referenced by open orders in `reserve_n_tags's exclusion set, or auto-close the order when` `set_keeper_close_flags` zeroes its last tagged balance (eliminates the wraparound variant).

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *If someone wants to create 30k orders, we should not prevent this*
>
> — Project0 team

Go back to Findings Summary

M16: Emissions config changes retroactively reprice already-accrued history

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/state/marginfi_account.rs, programs/marginfi/src/instructions/marginfi_group/configure_bank.rs	Type:	Logic error

Description

`claim_emissions` applies the bank's **current** `emissions_rate` and flag state to the **entire** unsettled interval `current_timestamp - balance.last_update`. Both `lending_pool_setup_emissions` and `lending_pool_update_emissions_parameters` overwrite the bank's rate and flags in place without:

- settling existing balances before the change;
- recording a per-bank epoch boundary or historical index;
- clamping `last_update` forward on affected balances.

On the next claim, the new configuration silently reprices historical time that was earned under the old configuration.

The direction of the drift depends on the admin action:

- **Over-distribution** — admin raises the rate, so unsettled balances get the new (higher) rate applied to the old (lower-rate) interval. `bank.emissions_remaining` is drained faster than intended, and later users receive capped or zero rewards.

- **Under-distribution** — admin lowers the rate, so unsettled users realize the low-rate tariff on time they earned at the higher rate, with no recourse after `last_update` advances.
- **Disable/re-enable round-trip** — a window that was paused is silently rewarded at the post-re-enable rate.

No malicious user action is required; any admin rate/flag change while unsettled balances exist triggers the drift.

Exploit scenario

1. Alice has an emissions-eligible deposit accruing silently at rate `R_old` and has not settled in a week.
2. Bob, the admin, calls `lending_pool_update_emissions_parameters` to raise the rate to `R_new = 10 * R_old` for a new promotion.
3. Alice immediately calls `claim_emissions`; the handler computes her reward as `R_new * (now - balance.last_update)`, crediting her for a week at ten times the rate she actually earned.
4. `bank.emissions_remaining` is depleted; users who deposit after the promotion receive capped or zero rewards.

Recommendation

Either settle all emissions-eligible balances (sweep) before writing the new rate or flags, or store per-bank epoch boundaries and split the settlement period across config-version boundaries inside `claim_emissions`. Alternatively, pin `balance.last_update = max(balance.last_update, emissions_config_last_update)` in `claim_emissions` so balances cannot claim across a config change.

Fix 1.1

The issue was fixed in commit `f28865d`[14]. The `lending_pool_setup_emissions`

and `lending_pool_update_emissions_parameters` instructions were removed; the only remaining admin-side emissions instruction is `lending_pool_reclaim_emissions_vault`, which empties the vault and zeroes `emissions_rate`, `emissions_remaining`, `emissions_mint`, and the emission flags in a single non-reconfigurable wind-down. Retroactive repricing is no longer possible because there is no way to reconfigure emissions parameters anymore. The client added the following comment.

> *Emissions were removed*
>
> — Project0 team

Go back to Findings Summary

M17: Outbound vault transfers on Token-2022 transfer-fee mints underpay destinations and misaccount the outstanding counters

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/marginfi_group/{collect_bank_fees,configure_bank}.rs, programs/marginfi/src/instructions/marginfi_account/emissions.rs	Type:	Logic error

Description

Inbound transfers on Token-2022 `TransferFeeConfig` mints are grossed up (e.g. `calculate_pre_fee_spl_deposit_amount` in emissions setup and the deposit path), so vaults receive the intended nominal amount. The outbound side is asymmetric: several handlers debit the nominal `amount` from the source vault and pass the same `amount` to `transfer_checked`, so the destination receives `amount - fee`, the outstanding counter is decremented as if the full amount was delivered, and the delta is captured by the mint's withheld-fee mechanism.

Two concrete sites exhibit this pattern:

1. `lending_pool_collect_bank_fees` — underpay across all three destinations. Each category decrements `collected_*_outstanding` by the transfer's nominal `amount`. For a `TransferFeeConfig` mint, `fee_vault`, `insurance_vault`, and the external program `fee_ata` (to `global_fee_wallet`) each receive `amount - fee`. The counter zeroes before destinations receive the full amount the interest

model charged borrowers; the delta accrues silently to the mint authority.

2. Emissions withdraw — underpay to users.
`lending_account_withdraw_emissions` and `lending_account_withdraw_emissions_permissionless` take the settled amount from `settle_emissions_and_get_transfer_amount()` and pass it directly to `transfer_checked`, then decrement `emissions_outstanding` by the full settled amount. On a transfer-fee mint the vault is debited the full amount but the user receives `amount - fee`. Every emissions withdraw on a transfer-fee mint permanently loses the fee portion; across users, the cumulative delta accrues to the mint authority.

Exploit scenario

Collect-bank-fees underpay:

1. The bank's mint is Token-2022 with `TransferFeeConfig` enabled.
2. Interest accrues; `collected_group_fees_outstanding`, `collected_insurance_fees_outstanding`, and `collected_program_fees_outstanding` grow.
3. Alice, a keeper, calls `lending_pool_collect_bank_fees`. Bob, the intended fee recipient (for example the `global_fee_wallet`), expects to receive the nominal fee amount.
4. Each SPL transfer sends `amount` from the liquidity vault; each destination ATA receives `amount - fee`.
5. The outstanding counters are decremented by the full pre-fee `amount`.
6. The fee portion accrues to the mint's withheld-fee mechanism. Repeating every collect cycle, the delta grows unboundedly.

Emissions-withdraw underpay:

1. The admin configures emissions on a Token-2022 mint with

`TransferFeeConfig` enabled.

2. Setup grosses up the inbound transfer, so the emissions vault holds the full intended `emissions_remaining`.
3. Alice calls `lending_account_withdraw_emissions` (or the permissionless variant) for a settled balance. Bob, the mint authority, captures withheld fees.
4. The handler transfers the full settled amount from the vault; Alice's ATA receives `amount - fee`.
5. `emissions_outstanding` is decremented as if Alice had received the full amount.
6. The fee portion is trapped in the mint's withheld-fee mechanism; across users, the cumulative delta accrues to the mint authority instead of the protocol or the user.

Recommendation

Fix the outbound side symmetrically with the inbound grossing-up. Read `post - pre` vault/destination balances around every outbound `transfer_checked` and decrement the outstanding counter by the actually-received amount; or gross up the transfer via `TransferFeeConfig::calculate_pre_fee_amount` so destinations receive the nominal amount with the extra cost paid by the source vault. Apply the same fix shape to all three destinations inside `lending_pool_collect_bank_fees` and to both `lending_account_withdraw_emissions` variants.

Partial solution 1.1

The issue was partially fixed in commit `f28865d`[15]. The emissions-side underpayment is gone because the `lending_account_withdraw_emissions[_permissionless]` instructions were removed entirely. However, `lending_pool_collect_bank_fees` still decrements

`collected_insurance_fees_outstanding`, `collected_group_fees_outstanding`, and `collected_program_fees_outstanding` by the pre-transfer-fee amount, so on Token-2022 mints with a transfer fee the destination still receives less than the bookkeeping reflects. The client relies on admin token vetting to ensure no fee-on-transfer mint is ever listed and added the following comment.

> *we are not supporting fee-on-transfer tokens, validated by admin*
>
> — Project0 team

Go back to Findings Summary

L1: Global `panic_pause` does not take effect until per-group propagation

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/panic_pause.rs	Type:	Logic error

Description

`panic_pause` updates only the global `FeeState.panic_state` but user-facing instructions gate on `group.is_protocol_paused()`, which reads exclusively from `MarginfiGroup.panic_state_cache`. The cache is updated only by the permissionless `propagate_fee_state` instruction (per-group).

Consequently, the emergency stop is not atomic: after `panic_pause` lands, user operations continue to succeed against any group whose cache has not yet been updated via `propagate_fee_state`.

This is distinct from W5 (new groups not initializing from fee state): here, even existing groups remain effectively unpaused until propagation runs.

Exploit scenario

1. Alice, the protocol admin, submits `panic_pause` during an active incident.
2. Bob, an attacker, continues calling pause-gated instructions (deposit/withdraw/borrow/repay/liquidate/etc.) against a group whose `panic_state_cache` is still unpaused.
3. Only after a separate `propagate_fee_state` transaction lands for that group do those instructions begin rejecting with `ProtocolPaused`.

MEV/front-running angle: even if a bot submits `propagate_fee_state` immediately after `panic_pause`, Bob can race a higher-priority transaction into the same block before the propagation transaction, extending the drain window.

Recommendation

Make pause/unpause effective without requiring a follow-up per-group instruction.

Options: - Update `panic_pause` to also update `MarginfiGroup.panic_state_cache` (requiring the group(s) be passed in and mutated) - Change pause guards to consult the authoritative `FeeState.panic_state` directly (and treat the cache as an optimization, not the sole source of truth)

Regardless of approach, the invariant must be: once `panic_pause` lands, pause-gated instructions must fail immediately for all groups in scope.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *In protocol procedures both should be called in same TX.*
>
> — Project0 team

Go back to Findings Summary

L2: Configuring rate limits resets limiter windows, allowing history erasure via no-op updates

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/configure_rate_limits.rs	Type:	Logic error

Description

`configure_bank_rate_limits` and `configure_group_rate_limits` call `configure_hourly` / `configure_daily`, which re-initialize the rate limit windows (resetting `window_start`, `prev_window_outflow`, and `cur_window_outflow`) each time they are invoked.

Therefore, an admin can erase accumulated outflow history by re-submitting the same limit values (a no-op config update). This undermines the intended safety property of rate limiting as an outflow throttle, especially under a compromised-admin threat model.

Exploit scenario

1. A group/bank has an active hourly/daily limiter and has already consumed most of its outflow budget.
2. Alice, a compromised admin, calls `configure_*_rate_limits(Some(current_value), None)` (or daily analog) without changing the cap.
3. The limiter window is reset, restoring the full outflow capacity immediately.
4. Bob, an attacker, repeats this periodically to allow sustained outflow

without ever hitting the configured limits.

Recommendation

Separate “update cap” from “reset window state”.

- Updating `max_outflow` must not reset `cur_window_outflow`, `prev_window_outflow`, or `window_start`.
- If an explicit “reset” is required operationally, provide a separate instruction with distinct intent, logging, and ideally tighter governance (timelock/guardian).

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *its by design - admin is trusted entity and multisig.*
>
> — Project0 team

Go back to Findings Summary

L3: Emode clone skips configuration validation

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/emode_clone.rs	Type:	Data validation

Description

The `lending_pool_clone_emode` instruction copies the entire `emode` configuration from one bank to another without running any post-copy validation on the destination bank.

In contrast, direct emode configuration path validates the resulting configuration before persisting it. The clone path therefore creates an inconsistency in safety checks and can apply parameter combinations that are not re-verified in this instruction.

Listing 3. Excerpt from emode_clone

```
6  pub fn lending_pool_clone_emode(ctx: Context<LendingPoolCloneEmode>) ->
   MarginfiResult {
7      let group = ctx.accounts.group.load()?;
8
9      check!(
10         ctx.accounts.signer.key() == group.admin || ctx.accounts.signer.key()
   == group.emode_admin,
11         MarginfiError::Unauthorized
12     );
13
14     let source_bank = ctx.accounts.copy_from_bank.load()?;
15     let mut destination_bank = ctx.accounts.copy_to_bank.load_mut()?;
16
17     destination_bank.emode = source_bank.emode;
18
19     msg!(
20         "emode settings copied from {:?} to {:?}",
```

```
        ctx.accounts.copy_from_bank.key(),
        ctx.accounts.copy_to_bank.key()
    );

    Ok(())
}
```

Exploit scenario

1. Alice, the `group.emode_admin`, configures a source bank with emode settings that are valid in the source context.
2. Alice calls the `lending_pool_clone_emode` instruction to copy those settings to Bob's destination bank.
3. The instruction writes `destination_bank.emode` directly without validating the resulting destination configuration.
4. Bob, a user of the destination bank, is exposed to behavior that bypassed the expected validation checkpoint for emode changes.

Recommendation

Run the same emode validation checks in the clone path before persisting copied settings.

Acknowledgment 1.1

The client acknowledged the finding and provided the following comment:

> *It is validated on the multisig level, the same as adding a new bank.*
>
> — Project0 team

Go back to Findings Summary

L4: Isolated assets can be stranded after equity-based bankruptcy disablement

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/handle_bankruptcy.rs	Type:	Logic error

Description

In the equity bankruptcy path, isolated assets are treated as zero. Bankruptcy is then checked using that equity value.

If an account still holds isolated-token value but other collateral drops, the account can be marked bankrupt earlier than intended because isolated assets are not counted in equity.

When `lending_pool_handle_bankruptcy` runs, it sets `ACCOUNT_DISABLED` on the whole account. Disabled accounts cannot use normal withdraw instructions, so remaining isolated-token value can become stranded after a wrongful bankruptcy transition.

Listing 4. Excerpt from handle_bankruptcy

```
196 marginfi_account.set_flag(ACCOUNT_DISABLED, true);
```

Exploit scenario

1. Alice, a user, holds a regular collateral asset and an isolated asset, and has liabilities.
2. The regular collateral value drops while the isolated asset still has non-zero value.

3. Bob, a risk admin, executes the `lending_pool_handle_bankruptcy` instruction; the equity-based bankruptcy check can ignore isolated asset value because isolated assets are treated as zero.
4. The account is marked `ACCOUNT_DISABLED`, and Alice cannot use normal withdraw instructions to recover the remaining isolated tokens.

Recommendation

Allow isolated-asset recovery after account disablement.

Acknowledgment 1.1

The client acknowledged the finding and provided the following comment:

> *Bankruptcy handling is done by the risk admin, so it holds the assumption that the risk admin is trusted. It is usually called in a crisis situation, so we need to have control over it.*
>
> — Project0 team

Go back to Findings Summary

L5: `pulse_bank_price_cache` is not gated by the protocol pause

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/pulse_bank_price_cache.rs	Type:	Access control

Description

During a protocol pause, operators generally expect bank state to be frozen. `pulse_bank_price_cache` can still mutate `bank.cache.last_oracle_price` and invoke `apply_untracked_inflow` (which updates the group rate limiter). An attacker preparing to liquidate at pause-end can pre-stage the cached price by repeatedly cranking `pulse` during the window.

Exploit scenario

1. The admin activates `panic_pause`.
2. Alice, preparing to liquidate Bob the moment the pause lifts, repeatedly calls `pulse_bank_price_cache`, steering `bank.cache.last_oracle_price` to a value favorable to her liquidation.
3. The pause lifts.
4. Alice's liquidation runs against the pre-staged cached price rather than a fresh post-pause observation.

Recommendation

Add `constraint = !group.load()?.is_protocol_paused() @ MarginfiError::ProtocolPaused` on `group`, consistent with the rest of the ops family.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *this is just cache for frontend and backend, it cannot create harm*
>
> — Project0 team

Go back to Findings Summary

L6: `close_balance` does not update bank total shares or position counts

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	programs/marginfi/src/state/marginfi_account.rs	Type:	Logic error

Description

`BankAccountWrapper::close_balance()` zeroes a user's `Balance` struct via `balance.close(true)` but does not:

1. Call `bank.change_asset_shares(-shares)` or `bank.change_liability_shares(-shares)` to remove dust shares from the bank's totals.
2. Call `bank.decrement_lending_position_count()` or `bank.decrement_borrowing_position_count()` to update position counters.

Compare with `withdraw_all()` and `repay_all()`, which both properly decrement bank totals and position counts before closing the balance.

`close_balance` succeeds when both asset and liability amounts are below `ZERO_AMOUNT_THRESHOLD` (0.0001 tokens). Each call leaks up to that amount of phantom shares into the bank's `total_asset_shares` / `total_liability_shares`, and leaves `lending_position_count` / `borrowing_position_count` permanently inflated by 1.

Consequences:

- **Bank closure blocked** — `lending_pool_close_bank` requires `lending_position_count == 0 && borrowing_position_count == 0`. A bank where many users called `close_balance` instead of `withdraw_all` / `repay_all` may become impossible to close because position counts are stuck above

zero with no remaining user balances to decrement them.

- **Phantom share accumulation** — over many calls, the bank's total shares drift upward relative to the actual sum of user balances, slightly distorting utilization ratio and interest rate calculations.

Exploit scenario

1. Alice and other users routinely call `close_balance` (instead of `withdraw_all` / `repay_all`) to tidy up dust positions on bank `B`.
2. Each call leaves `B.lending_position_count` or `B.borrowing_position_count` inflated by one without decrementing bank totals.
3. Bob, the admin, tries to decommission `B` via `lending_pool_close_bank`; the close reverts because the counters are non-zero.
4. No remaining user balances exist to withdraw through and decrement the counters; `B` becomes permanently un-closable via the normal path.

Recommendation

In `close_balance()`, before calling `balance.close(true)`, decrement the bank's totals and position counts to mirror the logic in `withdraw_all()` and `repay_all()`. Specifically: if the balance had assets, invoke `bank.change_asset_shares` with the negated `asset_shares` and then call `bank.decrement_lending_position_count()`; if it had liabilities, invoke `bank.change_liability_shares` with the negated `liability_shares` and then call `bank.decrement_borrowing_position_count()`.

Fix 1.1

The issue was fixed in commit `186b248`[16]. `BankAccountWrapper::close_balance()` now unwinds the closing balance's `asset_shares` / `liability_shares` from `bank.total_*_shares` and updates `lending_position_count` / `borrowing_position_count` using the same threshold

semantics as the internal accounting paths (avoiding double-decrements), preventing phantom share drift and stuck non-zero position counters that could block bank closure.

Go back to Findings Summary

L7: `transfer_to_new_account` state migration is incomplete — missing field copies, orphan `LiquidationRecord`, and permanently un-closable old account

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/marginfi_account/transfer_account.rs, programs/marginfi/src/instructions/marginfi_account/close.rs, programs/marginfi/src/state/marginfi_account.rs	Type:	Logic error

Description

`transfer_to_new_account` (and its PDA variant) migrates an account's authority by copying selected fields onto a fresh `MarginfiAccount` and disabling the old one. The migration is incomplete at multiple points, and the downstream close path inherits the problem.

Fields not copied / not rebound on migration. The migration copies `lending_account`, `emissions_destination_account`, and `account_flags`, but not `health_cache` (which stays zeroed from `load_init`) and not `liquidation_record` (which stays `Pubkey::default()`). The old account's `liquidation_record` is not cleared either, leaving it orphaned. A liquidator acting on the new key will miss historical entries; any component that relies on `health_cache` as a fast-path hint reads stale zeros and is forced onto a longer recompute path.

Old account is permanently un-closable. After migration the old account is

`ACCOUNT_DISABLED`, and `can_be_closed()` returns false whenever `ACCOUNT_DISABLED` is set. `marginfi_account_close` therefore can never close the old account. Every migration permanently strands ~0.018 SOL of rent. Combined with the orphan `LiquidationRecord` above, the stranded rent is larger still.

Both sites share the same class — migration-time hygiene on `transfer_to_new_account` — and a single review of the migration/close pair resolves them together.

Exploit scenario

1. Alice transfers her marginfi account to Bob via `transfer_to_new_account` (or the PDA variant).
2. Bob's new account starts with a zeroed `health_cache` and `liquidation_record == Pubkey::default()`; Alice's `liquidation_record` is left on-chain, orphaned and unreachable from Bob's new key.
3. A liquidator that relies on `health_cache` reads stale zeros on the new account and routes off a longer recompute path; historical liquidation entries are not available from Bob's key either.
4. Alice then calls `marginfi_account_close` on the old key to reclaim rent. `can_be_closed()` returns false because of the `ACCOUNT_DISABLED` flag, and the close reverts. ~0.018 SOL of rent is permanently stranded in the old PDA, and the orphan `LiquidationRecord` adds its own stranded rent on top.

Recommendation

Close the migration on both sides. In `transfer_to_new_account` (both variants), either copy `health_cache` (or explicitly re-invalidate it with `engine_ok = false`) and rebind or re-initialize a fresh `liquidation_record` on the new account, or document explicitly that history is intentionally discarded on migration. In `marginfi_account_close`, allow closing accounts that are `ACCOUNT_DISABLED &&`

`migrated_to != Pubkey::default()`, reclaiming rent to the authority (or original `fee_payer`), and extend the close instruction to also close the associated orphan `LiquidationRecord`.

Partial solution 1.1

The issue was partially fixed in commit `390dae2`[17]. The orphan `LiquidationRecord` portion is resolved: a new permissionless `close_liquidation_record` instruction returns rent to the original `record_payer` after a 60-day inactivity period, and `marginfi_account_close` now requires `liquidation_record == Pubkey::default()` and `active_orders == 0` before closing. However, the migration-hygiene gaps remain unaddressed: `health_cache` is still not copied or explicitly invalidated on transfer, and the old account remains permanently un-closable because `can_be_closed()` still rejects any `ACCOUNT_DISABLED` account — the recommended `ACCOUNT_DISABLED && migrated_to != Pubkey::default()` carve-out was not added, so the old account's rent (~0.018 SOL) is still stranded after every migration. The client added the following comment.

> *PR fixes issue: 1) liquidation record can be closed even if account was transferred*
>
> — Project0 team

The client also acknowledged the unresolved portion of the issue with the following comment.

> - *its a bit easier for us to index such transfer when both accounts (pre and post migration) still exists*
> - *if someone transfer, then close and reopen it can led to some weird scenarios with off-chain systems*
>
> *so these edge cases are not worth this little rent,*

> *especially as account transfers are not common at all.*
>
> — Project0 team

Go back to Findings Summary

L8: `deposit_up_to_limit` uses pre-accrual capacity, then fails post-accrual

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/marginfi_account/deposit.rs	Type:	Logic error

Description

`deposit_up_to_limit` computes the deposit amount against capacity **before** `accrue_interest` runs:

```
let deposit_amount = if deposit_up_to_limit {
    amount.min(bank.get_remaining_deposit_capacity()?)
} else {
    amount
};
if deposit_amount == 0 { return Ok(()); }
bank.accrue_interest(...)?;
```

`accrue_interest` then increases `total_asset_amount`, which shrinks the true capacity, and `change_asset_shares` can revert with `BankAssetCapacityExceeded` — defeating the idempotent-fit contract of `deposit_up_to_limit`.

Exploit scenario

1. Alice calls `deposit_up_to_limit` near the bank's current cap. Bob, a user, expects `deposit_up_to_limit` to deposit "as much as possible" without reverting.
2. The handler reads pre-accrual capacity and clamps `amount` to that number.
3. `accrue_interest` raises `total_asset_amount` enough that the clamped

deposit now exceeds capacity.

4. `change_asset_shares` reverts with `BankAssetCapacityExceeded`; Alice's "fit whatever you can" call fails entirely instead of depositing a slightly smaller amount.

Recommendation

Compute `deposit_amount` after `accrue_interest`.

Fix 1.1

The issue was fixed in commit `984cefa`[18]. `bank.accrue_interest` is now called before `get_remaining_deposit_capacity` in `lending_account_deposit`, so the capacity check is performed against post-accrual state and matches the same state used by the subsequent share-mint math.

Go back to Findings Summary

L9: `should_skip_rate_limit` misses `ACCOUNT_IN_ORDER_EXECUTION`

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	programs/marginfi/src/state/rate_limiter.rs	Type:	Logic error

Description

The predicate lists three bypass flags but not the order-execution flag:

```
pub fn should_skip_rate_limit(account_flags: u64) -> bool {
    (account_flags & ACCOUNT_IN_FLASHLOAN) != 0
        || (account_flags & ACCOUNT_IN_RECEIVERSHIP) != 0
        || (account_flags & ACCOUNT_IN_DELEVERAGE) != 0
}
```

`start_execute_order` sets `ACCOUNT_IN_ORDER_EXECUTION` and then performs deposit, withdraw, borrow, and repay moves. Those consume bank- and group-level rate-limit budget inside the bracketed transaction, potentially exhausting the daily budget for other users or failing the order mid-execution.

Exploit scenario

1. Group-level rate-limit budget is tight, and Bob submits a large order that triggers `start_execute_order`.
2. A keeper calls `start_execute_order` on Bob's order; the handler sets `ACCOUNT_IN_ORDER_EXECUTION`.
3. The limiter does not short-circuit for that flag, so the in-order withdraw consumes the remaining daily capacity.

4. The withdraw fails mid-flight, aborting the order while fees have already been paid and tags marked; Alice's unrelated withdraws can also fail due to the consumed budget.

Recommendation

Add `|| (account_flags & ACCOUNT_IN_ORDER_EXECUTION) != 0`, matching how order execution is treated symmetrically with flashloan elsewhere in the protocol.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *we don't like situation in which someone can withdraw tokens with fake orders*
>
> — Project0 team

Go back to Findings Summary

L10: `lending_account_withdraw_emissions_permissionless` has no session-flag gate

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_account/emissions.rs	Type:	Access control

Description

The permissionless variant blocks `ACCOUNT_DISABLED` and `ACCOUNT_FROZEN` in the handler but does not block `ACCOUNT_IN_FLASHLOAN`, `ACCOUNT_IN_ORDER_EXECUTION`, `ACCOUNT_IN_RECEIVERSHIP`, or `ACCOUNT_IN_DELEVERAGE`. During receivership in particular, a parallel permissionless token CPI violates the exclusive-control assumption held by the receiver.

Exploit scenario

1. A receivership session is active on Alice's account.
2. Bob, a keeper, calls `lending_account_withdraw_emissions_permissionless`; the session-flag gate is absent.
3. The CPI transfers emissions to the stored destination wallet while the receiver is mid-process.
4. The receiver's exclusive-control assumption is violated, complicating state reconciliation.

Recommendation

Reject `ACCOUNT_IN_FLASHLOAN | ACCOUNT_IN_ORDER_EXECUTION | ACCOUNT_IN_RECEIVERSHIP | ACCOUNT_IN_DELEVERAGE`, matching `place_order` and

`start_execute_order`.

Fix 1.1

The issue was fixed in commit `f28865d`[19]. The `lending_account_withdraw_emissions_permissionless` instruction (and its authority-gated sibling `lending_account_withdraw_emissions`) was removed entirely as part of the emissions wind-down, so there is no longer any path that bypasses session flags through emissions withdrawal. The client added the following comment.

> *`emissions were removed`*
>
> — Project0 team

Go back to Findings Summary

L11: `marginfi_account_update_emissions_destination_account` does not validate the stored destination's type

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/marginfi_account/emissions.rs	Type:	Data validation

Description

`destination_account: AccountInfo` in the update instruction is unchecked; its key is stored raw. The permissionless withdraw later treats this key as a **wallet** (owner of an ATA). Users who store a token account (ATA or SPL) instead of a wallet, or a non-signing PDA, permanently brick the permissionless withdraw path until they call update again.

`place_order` also requires `emissions_destination_account != Pubkey::default()` as a precondition, so a misconfiguration silently blocks order placement with no indication of the root cause.

Exploit scenario

1. Following a UI that prompts for "destination account", Alice copies her ATA address into `marginfi_account_update_emissions_destination_account`.
2. Alice places an order; the `emissions_destination_account != Pubkey::default()` precondition is satisfied.
3. Bob calls the permissionless withdraw; it derives an ATA from Alice's ATA address, producing an unrelated (and non-existent) destination.

4. Emissions never flow, and the UI provides no diagnostic.

Recommendation

Deserialize as `InterfaceAccount<'info, TokenAccount>` and store `owner` instead, or rename the field and clearly document in IDL and UI that this must be a wallet.

Fix 1.1

The issue was fixed in commit `f28865d`[20]. The `marginfi_account_update_emissions_destination_account` instruction is still present so users can record an off-chain destination, but every on-chain consumer that routed tokens to that field (`lending_account_withdraw_emissions[_permissionless]`) was removed and `place_order` no longer requires the field to be non-default, so a mistyped destination can no longer brick any on-chain flow. The client added the following comment.

> *emissions were removed, this function is still there so users can dictate*
>
> — Project0 team

Go back to Findings Summary

L12: Fee state propagation has no version or consistency check between `FeeState` and `group.fee_state_cache`

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/propagate_fee_state.rs, programs/marginfi/src/instructions/marginfi_group/config_group_fee.rs, programs/marginfi/src/state/marginfi_group.rs	Type:	Front-running

Description

`propagate_fee` copies `program_fee_fixed` / `program_fee_rate` and panic state from `FeeState` into `group.fee_state_cache`, but there is no version counter or staleness check. After `edit_global_fee_state`, each group remains at the old rate until `propagate_fee` is run for it. `get_fees()` gates on `group.group_flags.PROGRAM_FEES_ENABLED` rather than on the live `FeeState` rate, so there is also no atomic disable.

Rate-change timing becomes attacker-selectable: a borrower who knows rates are about to fall can crank accrual before propagation, and an attacker who knows rates are about to rise can race `propagate_fee` and then `accrue_interest` to lock in the old rate.

Exploit scenario

Rate-fall scenario:

1. Alice, the global-fee admin, lowers `program_fee_rate` in `FeeState`.
2. Bob, a depositor who prefers the old (higher) rate, cranks `accrue_interest` on his bank before anyone calls `propagate_fee` for his group.
3. Accrual runs at the stale (higher) cached rate, locking in a higher yield for Bob.

Rate-rise scenario:

1. Alice raises `program_fee_rate` in `FeeState`.
2. Bob, a borrower, races `propagate_fee` on his own group and then `accrue_interest` ahead of other users in other groups.
3. Other groups continue to accrue at the pre-rise rate until their propagation runs, while Bob benefits from selective timing across groups.

Recommendation

Add a monotonic `version: u64` to `FeeState` that bumps on every edit; mirror it in `fee_state_cache`; require equality (or a small bounded drift) at accrual time. Alternatively, pass `FeeState` directly into `accrue_interest` so there is one source of truth.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *we just propagate when change fee, this is the internal process*
>
> — Project0 team

Go back to Findings Summary

L13: `lending_pool_withdraw_fees_permissionless` forwards direct vault donations

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/collect_bank_fees.rs	Type:	Logic error

Description

`amount` is clamped only to `fee_vault.amount`. If a third party accidentally or adversarially transfers tokens of the bank's mint directly into the `fee_vault` PDA, the next permissionless crank forwards them to `fees_destination_account` — which may be an account the original sender does not control (see L19).

Exploit scenario

1. Alice accidentally sends USDC directly to a bank's `fee_vault` PDA instead of the intended recipient.
2. Bob, the next caller of `lending_pool_withdraw_fees_permissionless`, sweeps the full `fee_vault` balance — including Alice's donation — into the current `fees_destination_account`.
3. Alice has no path to recover the mistakenly-sent tokens.

Recommendation

Track an in-bank `collected_group_fees_pending_permissionless_withdraw` counter and clamp `amount` to it, so unsolicited deposits remain recoverable through an admin-only path.

Fix 1.1

The issue was fixed in commit `f28865d`[21] as part of the broader fee / emissions cleanup. The client added the following comment.

> *emissions were removed*
>
> — Project0 team

Go back to Findings Summary

L14: `juplend_init_position` does not refresh rate or verify fToken mint, yet activates the bank

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/marginfi/src/instructions/juplend/init_position.rs	Type:	Data validation

Description

`juplend_init_position` transitions `operational_state` from `Paused` to `Operational` after a CPI deposit. It does not call `cpi_update_rate` first, does not read the pre/post fToken vault balance to verify minted shares, and does not assert a minimum. Any silent CPI under-delivery (e.g. the mint returning fewer than expected fTokens due to a stale rate) still flips the bank live.

Exploit scenario

1. JupLend's `Lending` account has not been refreshed for many slots; `token_exchange_price` is stale.
2. Bob, an attacker, calls `juplend_init_position` for the newly added, still-paused bank.
3. The CPI deposit mints fewer fTokens than the refreshed rate would produce.
4. The handler does not read the pre/post fToken-vault balance and does not enforce a minimum.
5. `bank.config.operational_state` flips from `Paused` to `Operational`.
6. Alice and every subsequent depositor interact with a live bank whose share accounting is initialized against an undercapitalized seed, skewing per-share values.

Recommendation

Call `cpi_update_rate` first; record the pre/post fToken-vault balance and assert `minted == expected` (or at least `> 0`). Only set `operational_state = Operational` after the mint verification succeeds.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *we have this process internally to run both transactions at once with script*
>
> — Project0 team

Go back to Findings Summary

L15: JupLend `claim_account` is unchecked and trusted to be unused by the CPI

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/marginfi/src/instructions/juplend/withdraw.rs	Type:	Trust model

Description

The `claim_account` on `JuplendWithdraw` carries no validation:

```
/// CHECK: not validated by JupLend - any mutable account works
#[account(mut)]
pub claim_account: UncheckedAccount<'info>,
```

The inline comment acknowledges this is a temporary trust assumption. Because marginfi signs the CPI with the `liquidity_vault_authority` PDA, JupLend treats whatever is passed as authorized for that signer. If a future JupLend upgrade reads or writes this account (e.g. as a rewards-claim destination), an attacker can redirect reward flows or grief the withdraw by passing a writable account that panics on write.

Exploit scenario

1. A future JupLend upgrade repurposes `claim_account` as a rewards destination (or any other side effect on write). Bob, a protocol user, expects rewards to flow according to JupLend's intended semantics.
2. Alice calls `juplend_withdraw` and supplies a `claim_account` she controls.
3. marginfi signs the CPI with the `liquidity_vault_authority` PDA, so JupLend treats Alice's account as authorized for that signer.

4. Rewards earmarked for the protocol flow to Alice.
5. Alternatively, Alice passes a writable account that panics on write, bricking `juplend_withdraw` for all users routed through that tx shape.

Recommendation

Either derive and enforce the canonical PDA (`seeds = [b"user_claim", liquidity_vault_authority.key().as_ref(), mint.key().as_ref()]`, `seeds::program = liquidity_program.key()`), or store the expected `claim_account` on `Bank` at `add_bank_juplend` / `init_position` time and enforce via `has_one`.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *we are waiting to juplend to clarify situation with this account (it should be optional) - there is comment in the code already*
>
> — Project0 team

Go back to Findings Summary

L16: Tokenless-deleveraging banks still contribute full Initial-margin collateral

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/marginfi/src/state/marginfi_account.rs, programs/marginfi/src/state/bank.rs, programs/marginfi/src/instructions/marginfi_account/repay.rs	Type:	Logic error

Description

When a bank enters the tokenless-deleveraging flow, the risk admin sets `TOKENLESS_REPAYMENTS_ALLOWED` (and later `TOKENLESS_REPAYMENTS_COMPLETE` auto-flips in `lending_account_repay` or is forced via `lending_pool_force_tokenless_repay_complete`). Tokenless repay discharges `liability_shares` with no token inflow, so the bank's vault no longer fully backs `total_asset_shares * asset_share_value`. Lenders' on-chain `asset_shares` keep their full nominal value until `lending_account_purge_delev_balance` wipes them.

The health engine has no branch to discount that phantom collateral for the Initial-margin requirement. Both `calc_weighted_asset_value_standalone` and `calc_weighted_asset_value_cached_standalone` in `programs/marginfi/src/state/marginfi_account.rs` only zero the Initial weight when `bank.config.operational_state == BankOperationalState::ReduceOnly` — the tokenless flags are ignored. An existing borrow-side guard (`programs/marginfi/src/instructions/marginfi_account/borrow.rs:295-297`)

blocks borrowing **from** a flagged bank and its in-code comment explicitly names the "admin forgot to put a deleveraging bank into reduce-only mode" footgun, but nothing blocks borrowing **against** it on another bank.

The result is that a lender on a tokenless bank B can keep pricing their under-collateralized B deposit at full oracle price times `asset_weight_init` while borrowing against it on an unrelated healthy bank G. Once purge or the natural auto-flip wipes the B asset_shares (no health check runs on purge), the account is left holding only the G liability, which eventually socializes onto G's lenders through `lending_pool_handle_bankruptcy`.

Exploit scenario

Preconditions: the risk admin has enabled `TOKENLESS_REPAYMENTS_ALLOWED` on bank B without also setting `BankOperationalState::ReduceOnly`, and Alice already holds a deposit in B from before the flag was set.

1. Alice leaves her B deposit untouched so the health engine keeps pricing it at full oracle price times `asset_weight_init`.
2. Alice calls `lending_account_borrow` against healthy bank G for the maximum Initial-margin-permitted amount, using the inflated B collateral.
3. Alice calls `lending_account_withdraw` to move the borrowed tokens from G to an external wallet.
4. `lending_pool_force_tokenless_repay_complete` runs (or the flag auto-flips) and `lending_account_purge_delev_balance` zeroes Alice's B asset_shares with no health check.
5. Alice walks away from the now-zero-collateral marginfi account; its G liability is written off via `lending_pool_handle_bankruptcy`, socializing the loss onto G's lenders.

Alice's profit per account is roughly `nominal_A_deposit * asset_weight_init_A - nominal_A_deposit * (vault_balance_A / total_nominal_deposits_A)` — the

full-value loan extracted from G minus the pro-rata recovery she would have received from A's drained vault. Aggregated across all pre-flag depositors running the same play, the upper bound on bad debt socialized to other banks is `(total_A_deposits * asset_weight_init_A) - vault_balance_A`.

Recommendation

Extend the Initial-margin zero-out in both `calc_weighted_asset_value_standalone` and `calc_weighted_asset_value_cached_standalone` to fire when the bank has `TOKENLESS_REPAYMENTS_ALLOWED` or `TOKENLESS_REPAYMENTS_COMPLETE` set, mirroring the existing `ReduceOnly` branch. This closes the same admin-forgot-ReduceOnly footgun that the `borrow.rs` constraint only partially addresses.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *codebase trusts risk admin, acknowledges this is a known footgun*
>
> — Project0 team

Go back to Findings Summary

L17: `lending_account_close_balance` does not check session flags

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_account/close_balance.rs	Type:	Access control

Description

`LendingAccountCloseBalance` enforces `ACCOUNT_FROZEN` and `ACCOUNT_DISABLED` via `account_not_frozen_for_authority` and `is_signer_authorized(…, false, false)`, but has no gate on `ACCOUNT_IN_FLASHLOAN`, `ACCOUNT_IN_ORDER_EXECUTION`, `ACCOUNT_IN_RECEIVERSHIP`, or `ACCOUNT_IN_DELEVERAGE`. The handler also calls `claim_emissions` and mutates `balance.last_update`, `bank.emissions_remaining`, and bank accounting — all while a session flag may be active.

Because `check_health_and_verify_unchanged` (order execution) and the end-flashloan health check only look at `shares`, this does not directly break either invariant today; but the mutation is inconsistent with every other mutating user-facing ix and broadens the composition surface (e.g. a future health check that reads `emissions_outstanding` would be violated). Combined with the bank-totals gap in `close_balance` (see L6), the blast radius widens if the call lands inside a flashloan.

Exploit scenario

1. Alice starts a flashloan, setting `ACCOUNT_IN_FLASHLOAN`.
2. Alice calls `lending_account_close_balance` on a dust balance inside the flashloan.

3. The handler calls `claim_emissions`, debiting `bank.emissions_remaining` and mutating `balance.last_update` while the flashloan invariant is live.
4. The end-flashloan check only compares `shares` and accepts the transaction.
5. Bob, a downstream consumer that reads `emissions_outstanding` or bank accounting, now sees state that diverges from the pre-flashloan snapshot.

Recommendation

Reject `ACCOUNT_IN_FLASHLOAN | ACCOUNT_IN_RECEIVERSHIP | ACCOUNT_IN_ORDER_EXECUTION | ACCOUNT_IN_DELEVERAGE` in the account constraints, matching deposit/withdraw/borrow/repay.

Fix 1.1

The issue was fixed in commit `f28865d`[22]. The concrete state-inconsistency vector is eliminated because `Balance::close` no longer calls `claim_emissions` and `lending_account_close_balance` no longer mutates emissions state, so the dangerous side effect that motivated the finding is gone. The literal recommendation (rejecting `ACCOUNT_IN_FLASHLOAN | ACCOUNT_IN_RECEIVERSHIP | ACCOUNT_IN_ORDER_EXECUTION | ACCOUNT_IN_DELEVERAGE` in the account constraints) was not applied. The client added the following comment.

> *We should allow closing balance in ACCOUNT_IN_ORDER_EXECUTION, other are not applicable*
>
> — Project0 team

Go back to Findings Summary

L18: No bounds validation on fee parameters in `init_global_fee_state` / `edit_global_fee_state`

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/init_global_fee_state.rs, programs/marginfi/src/instructions/marginfi_group/edit_global_fee.rs	Type:	Data validation

Description

`program_fee_fixed`, `program_fee_rate`, `liquidation_max_fee`, `order_execution_max_fee`, `bank_init_flat_sol_fee`, `liquidation_flat_sol_fee`, and `order_init_flat_sol_fee` are all written without bounds checks. `WrappedI80F48` is signed, so negative values are accepted.

Several pathological values are consequential:

- A negative `program_fee_rate` flips interest math in `calc_interest_rate_accrual_state_changes`.
- An overly large `program_fee_rate` overflows share-value updates, causing `MathError` reverts — protocol accrual is then bricked until the admin edits the state again (and then propagates — see L12).
- An excessive `liquidation_max_fee` (e.g. `0.99`) wipes 99% of a liquidation victim's collateral.
- An out-of-range `order_execution_max_fee` either disables the cap or bricks TP execution: `end_execute_order` computes `max_fee_frac = 1 - order_execution_max_fee` and `allowed_diff = start_health * max_fee_frac`,

then requires `net >= allowed_diff`. Setting `order_execution_max_fee > 1` makes `allowed_diff` negative (cap silently disabled); setting it `< 0` makes TP execution unreachable.

Exploit scenario

1. Alice, a compromised or malicious global-fee admin, sets `liquidation_max_fee = 0.99`.
2. The next liquidation runs against Bob's account.
3. The liquidator charges up to 99% of Bob's collateral as a liquidation fee instead of the intended single-digit percent.
4. Symmetric scenarios exist for negative `program_fee_rate` (flipping accrual sign) and excessive `program_fee_rate` (bricking accrual via overflow).

Recommendation

Enforce sensible bounds on every fee parameter at write time — for example `0 <= program_fee_rate < MAX_RATE`, `0 <= liquidation_max_fee <= 0.1`, `0 <= order_execution_max_fee < 1`, and so on — returning `MarginfiError::InvalidConfig` on violation.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *it's controlled by multisig, validation is on MS level*
>
> — Project0 team

Go back to Findings Summary

L19: `lending_pool_update_fees_destination_account` has no owner/state check on the destination

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/collect_bank_fees.rs	Type:	Data validation

Description

`destination_account: InterfaceAccount<TokenAccount>` in `LendingPoolUpdateFeesDestinationAccount` is constrained only by `mint == bank.mint` — there is no `owner` check and no `state == Initialized` check. A malicious or compromised admin can:

- Set the destination to a frozen token account, after which permissionless fee withdrawals always revert, forcing all fee flow through the admin path.
- Set it to a third-party account, effectively donating fees without triggering obvious admin-to-self flows.
- Set it to the bank's own `fee_vault`, `insurance_vault`, or `liquidity_vault` PDAs, creating no-op self-transfers or accounting confusion.

Exploit scenario

1. Alice, a compromised admin, calls `lending_pool_update_fees_destination_account` pointing at a frozen token account she controls.
2. Bob calls the permissionless `lending_pool_withdraw_fees_permissionless` crank; it reverts at CPI time because the SPL transfer into a frozen account

fails.

3. All fee flow is forced through the admin-only path, concentrating control back on Alice.

Recommendation

Add `constraint = destination_account.state == AccountState::Initialized`, require `destination_account.owner == admin.key()` (or a protocol-controlled wallet), and reject the bank's own vault PDAs. Emit a structured event on every change.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *it's controlled by multisig, validation is on MS level*
>
> — Project0 team

Go back to Findings Summary

L20: `lending_pool_withdraw_insurance` accepts unbounded amount, with no accrue or minimum-buffer invariant

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/collect_bank_fees.rs	Type:	Access control

Description

`lending_pool_withdraw_insurance` has several defense-in-depth gaps:

1. `amount` is bounded only by `insurance_vault.amount` — the admin can fully drain insurance in a single transaction.
2. No `accrue_interest` or `collect_bank_fees` call runs beforehand, so `collected_insurance_fees_outstanding` sitting on the liquidity-vault side is not moved into the insurance vault first.
3. `dst_token_account` has no explicit `mint == bank.mint` constraint; runtime CPI will catch a mismatch, but this is inconsistent with `update_fees_destination_account` which does validate the mint.
4. There is no floor like "insurance must remain at least X% of liabilities". A compromised admin can empty the backstop immediately before a bankruptcy event they front-run.

Exploit scenario

1. Alice, a compromised admin, observes that Bob's large borrow position is about to tip into bankruptcy.

2. Alice calls `lending_pool_withdraw_insurance` with `amount = insurance_vault.amount`, draining the backstop entirely.
3. Bob's bankruptcy is triggered.
4. Without insurance to absorb the loss, the bankruptcy socializes directly onto depositors.

Recommendation

Clamp `amount` with a minimum-buffer invariant (e.g. insurance must remain at least a configured fraction of liabilities), accrue and collect before draining, add the `mint == bank.mint` constraint on `dst_token_account`, and add a timelock or signalled intent for large withdrawals.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *admin is trusted entity run by governance MS level*
>
> — Project0 team

Go back to Findings Summary

L21: Frozen account authorities can still `close_order` and `set_keeper_close_flags`

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	programs/marginfi/src/instructions/marginfi_account/order.rs	Type:	Access control

Description

Both `CloseOrder` and `SetKeeperCloseFlags` use only `has_one = authority` — they do not invoke `account_not_frozen_for_authority` or `is_signer_authorized`. A frozen user can therefore:

- Close their own orders, reclaiming rent to an attacker-chosen `fee_recipient` (see also M9).
- Mutate `lending_account.balances[..].tag` to detach orders from their original balances, enabling keeper-close later.

Other order entry points (`place_order`, `start_execute_order`) correctly reject frozen accounts. This pair is the outlier. Undermines the documented freeze semantics ("authority blocked from major actions; admin retains control"). It is not a direct drain — order execution itself is blocked for frozen accounts at `start_execute_order` — but a frozen user can still dismantle their order-level protection and invalidate the order/balance tag state.

Exploit scenario

1. Bob, the admin, freezes Alice's account during a remediation.
2. Alice calls `close_order` with `fee_recipient = alice_wallet`, reclaiming rent despite the freeze.

3. Alice calls `set_keeper_close_flags` to detach outstanding orders from their balances.
4. Alice hollows out her own order-level protections in spite of Bob's intended freeze.

Recommendation

Add the standard constraint pair used by the deposit and withdraw instructions to both `CloseOrder` and `SetKeeperCloseFlags`: an `account_not_frozen_for_authority` check against the loaded `marginfi_account` and the `authority` key, followed by an `is_signer_authorized` check against the loaded `marginfi_account`, the group `admin`, and the `authority` key — with both the receivership-allowed and order-execution-allowed flags passed as `false` — so frozen authorities are rejected on these paths as they already are on `place_order` and `start_execute_order`.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *it should be fine, this is just rent*
>
> — Project0 team

Go back to Findings Summary

L22: `keeper_close_order` can be bricked by reusing the closed `marginfi_account` address

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_account/order.rs	Type:	Denial of service

Description

`keeper_close_order` treats the unchecked `marginfi_account` as "closed" only when `owner == system_program::ID && data_is_empty()`. If the same pubkey has been reused as a non-empty account (owned by a different program, or system-allocated data), the handler falls through to manual deserialization and returns `InternalLogicError` on length or discriminator mismatch. It never verifies marginfi-program ownership before attempting the parse.

For non-PDA (keypair-based) accounts, a user can close their marginfi account and later recreate the same address as something else, bricking the permissionless keeper-close cleanup path for any lingering stale order PDAs bound to that address.

Exploit scenario

1. Alice closes her non-PDA marginfi account, leaving an open `Order` PDA.
2. Alice reuses the same keypair to allocate an account owned by a different program.
3. Bob, a keeper, calls `keeper_close_order` against the lingering `Order` PDA; the handler sees a non-system-owned, non-empty account and falls through to deserialization.

4. The call reverts on discriminator mismatch.
5. The stale order PDA's rent is permanently stuck.

Recommendation

In the fall-through branch, first check `marginfi_account_info.owner == &crate::ID`. If not, treat the account as "gone" (equivalent to the `system_program + empty` branch) rather than attempting to deserialize foreign bytes.

Fix 1.1

The issue was fixed in commit `a5fda8c`[23]. The fall-through deserialization path in `keeper_close_order` is unchanged, but `marginfi_account_close` now enforces `active_orders == 0` (and `liquidation_record == Pubkey::default()`) before closing, so the exploit precondition — closing a non-PDA marginfi account while leaving open `Order` PDAs behind — is no longer reachable. The client added the following comment.

> *we are checking now if there are orders before closing*
>
> — Project0 team

Go back to Findings Summary

L23: Shared rate limiter uses unchecked `u64` → `i64` narrowing

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/marginfi/src/state/rate_limiter.rs, type-crate/src/types/rate_limiter.rs, programs/marginfi/src/instructions/marginfi_group/configure_rate_limits.rs	Type:	Overflow/Underflow

Description

The limiter stores `max_outflow: u64` but accumulates `cur_window_outflow: i64` and `prev_window_outflow: i64`, then compares via `self.max_outflow as i64` and `amount as i64` — all unchecked narrowing casts. If the admin configures `max_outflow > i64::MAX`, or any amount exceeds `i64::MAX` (practically unreachable from user amounts but not from configured limits), the sign wraps negative. `remaining_capacity()` then returns a negative value, and the limiter starts making allow/deny decisions against corrupted state.

Consequences:

- Remaining capacity stops representing the real configured limit.
- Hourly or daily enforcement becomes inconsistent.
- Subsequent deposits and repays offset capacity based on already-corrupted state.
- A core safety control silently becomes broken — an admin who intends to "raise the limit to `u64::MAX`" actually bricks outflows.

Exploit scenario

1. Alice, the admin, intends to effectively disable the rate limiter and sets `max_outflow = u64::MAX`.
2. Inside the limiter, the cast `self.max_outflow as i64` wraps to `-1`.
3. `remaining_capacity()` returns a negative value, and every subsequent outflow is denied.
4. Alice's "open the taps" configuration silently bricks withdrawals for Bob and other users.

Recommendation

Enforce `max_outflow <= i64::MAX as u64` at configure time (returning `InvalidConfig`), or use checked / `try_from` conversions inside the limiter (`i64::try_from(amount)?`) and propagate an error on overflow.

Fix 1.1

The issue was fixed in commit `55ac554`[24]. The unchecked `as i64` narrowing is replaced with `i64::try_from`-based `amount_as_i64` and `clamp_i128_to_i64` helpers, and `configure_bank_rate_limits` / `configure_group_rate_limits` now gate inputs through `is_valid_rate_limit_amount`.

Go back to Findings Summary

L24: Bank onboarding accepts mints whose transfer rules break protocol vault assumptions

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/marginfi/src/instructions/marginfi_group/add_pool.rs, programs/marginfi/src/state/bank.rs, programs/marginfi/src/utils/general.rs	Type:	Data validation

Description

`lending_pool_add_bank` and the similar bank creation paths accept `bank_mint`, create `liquidity_vault`, `insurance_vault`, and `fee_vault` for that mint, save the mint in bank state, and then validate only `BankConfig`. They do not inspect the mint itself. For Token-2022, this means onboarding never checks whether the mint has extensions that change who can move tokens or whether normal vault transfers can work at all.

The same assumption continues at runtime. `maybe_take_bank_mint` only checks that the mint passed in `remaining_accounts` matches `bank.mint`. It does not inspect extensions. Later, token movement is delegated to `deposit_spl_transfer` and `withdraw_spl_transfer`, so SPL Token or Token-2022 decides whether the transfer succeeds and who is allowed to authorize it.

The mint can then break custody in several ways:

- `PermanentDelegate` lets an external key transfer or burn tokens from protocol vaults directly through Token-2022.

- `TransferHook` routes every vault transfer through external code that a live hook authority can later replace.
- A live SPL `freeze_authority` or Token-2022 `Pausable` lets the issuer freeze the vaults or pause the mint after the bank has accumulated user and protocol balances.
- `DefaultAccountState(Frozen)` and `NonTransferable` make the mint incompatible with ordinary vault transfers from the start.

The common cause is that onboarding never checks the mint's transfer rules against the vault custody model it is about to build.

Exploit scenario

1. Alice, a governance actor, lists a mint through one of the standard onboarding instructions (`lending_pool_add_bank`, `add_pool_with_seed`, or the Drift / Solend / Kamino / JupLend variants).
2. Bob, the mint issuer or a privileged mint authority, deploys a mint that carries one of the unsupported controls above, but onboarding accepts it because it validates only `BankConfig`.
3. The vaults are created and the bank is stored under the assumption that PDA ownership of the vault is sufficient.
4. Later, the mint's own rules take over: a permanent delegate can move or burn vault tokens directly; a mutable transfer hook can redirect or block transfers through external code; a live freeze or pause authority can disable a bank that already holds balances; and a frozen-by-default or non-transferable mint makes the bank unusable from creation.
5. Standard flows (deposit, repay, borrow, withdraw, fee collection, fee withdrawal, insurance handling) then fail at the token program boundary, and in the `PermanentDelegate` case are bypassed entirely while marginfi's own checks remain green.

Recommendation

Screen mint features in every bank creation flow before the vaults are created and the bank is stored:

- reject mints with `PermanentDelegate`, `Pausable`, `NonTransferable`, or `DefaultAccountState(Frozen)`;
- reject mints with a live `freeze_authority` unless governance explicitly accepts that risk and has matching operational controls;
- reject `TransferHook` unless the hook program is explicitly allowlisted and the hook update authority has been permanently disabled;
- verify that the new vault accounts will be transferable under the protocol's custody model before the asset is listed.

Where any of these features is intentionally supported, the exception must be per asset and recorded on the bank, not implicit in unrestricted onboarding.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *each listing is going through the manual review process*
>
> — Project0 team

Go back to Findings Summary

W1: `daily_limit = 0` disables deleverage withdrawal limit enforcement while setter rejects 0

Impact:	Warning	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_group/configure_withdrawal_limit.rs	Type:	Logic error

Description

`configure_deleverage_withdrawal_limit` rejects `daily_withdrawal_limit = 0`, but enforcement in `MarginfiGroup::update_withdrawn_equity` treats `daily_limit == 0` as "no limit" (skips the check).

This creates an unsafe default: newly initialized or upgraded groups that have not had the limit configured operate with no deleverage withdrawal throttling until the admin sets it.

Exploit scenario

1. Alice, a group admin, initializes or upgrades a group and leaves the deleverage withdrawal limit at its default `0`.
2. Bob uses deleverage flows; they record withdrawn equity but skip the limit check entirely because `daily_limit == 0`.
3. The intended throttle is absent until Alice performs an explicit configuration action.

Recommendation

Unify semantics: - Either allow setting `0` and document it as "disabled", while also ensuring defaults are set explicitly at initialization time, or - Treat `0` as "blocked / must configure" in the enforcement path (safer-by-default), so

the limit is enforced unless explicitly configured otherwise.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *risk admin is trusted.*
>
> — Project0 team

Go back to Findings Summary

W2: Group admin can withdraw emissions from frozen accounts to an arbitrary destination

Impact:	Warning	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_account/emissions.rs	Type:	Access control

Description

`lending_account_withdraw_emissions` allows the group admin to act on frozen accounts (via `is_signer_authorized`'s "admin allowed when frozen" branch). The instruction also accepts an arbitrary `destination_account: TokenAccount` provided by the caller.

As a result, if an account is frozen, the group admin can withdraw that account's accrued emissions and route them to any token account of their choosing, without using the user-configured `emissions_destination_account`.

This may be intended under a "seizure/remediation" model, but it is a concrete privilege that should be explicitly documented and should emit an auditable trail consistent with the protocol's governance assumptions.

Exploit scenario

1. Alice, the group admin, freezes Bob's user account.
2. Alice calls `lending_account_withdraw_emissions` with `destination_account` set to an admin-controlled token account.
3. Bob cannot prevent or redirect the transfer while frozen, and the transfer does not necessarily go to the user-configured emissions destination.

Recommendation

Choose and document the intended semantics: - If admin seizure is intended: add explicit documentation and emit a dedicated event including the emissions destination token account used. - If seizure is not intended: restrict `destination_account` to the canonical ATA of `marginfi_account.emissions_destination_account` (or disallow admin-signed withdrawals entirely).

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *this is necessary for compliance reasons.*
>
> — Project0 team

Go back to Findings Summary

W3: Outflow circuit breakers are controlled by the same admin they intend to constrain

Impact:	Warning	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_group/{configure_rate_limits,configure_withdrawal_limit,panic_pause}.rs	Type:	Trust model

Description

The protocol includes multiple “circuit breaker” style controls (panic pause, bank/group rate limits, deleverage withdrawal limit). However, these controls are configured and operated by privileged keys that are part of the same trust domain as the actions they are intended to throttle.

Under a compromised-admin threat model, these controls do not provide meaningful protection: the attacker can either disable them, reconfigure them, or reset their state to maintain throughput.

This is a protocol-level governance and threat-model issue rather than a single-instruction bug, but it is security-relevant because it affects the guarantees the system can claim during incident response.

Exploit scenario

1. Alice’s admin key is compromised.
2. Bob, an attacker, raises or disables configured throttles (or resets limiter state via config changes).
3. The advertised circuit breakers fail to slow or stop the outflow, despite being present in the codebase.

Recommendation

Introduce separation of duties for emergency and throttling controls: - Use a guardian / timelock / multisig distinct from the primary admin for (a) configuring rate limits and deleverage caps and (b) executing panic pause. - Where immediate response is required, prefer “guardian can only tighten / pause” semantics, with loosening only via timelock. At minimum, explicitly document that these controls are not designed to defend against a hostile/compromised admin.

Partial solution 1.1

The issue was partially fixed in the commit `5b8337d`[25] by adding a `pause_delegate_admin` that can trigger `panic_pause` but cannot unpause or change other fee settings, giving the panic circuit breaker tighten-only separation from `global_fee_admin`.

The client also commented on the issue with the following comment.

> *there is pause delegate admin that can act fast. we believe that other role for outflow setting is not necessary. for outflow setting our MS adds protection layer on top of that. ideally it should be security council and timelock. this is in-progress.*
>
> — Project0 team

Go back to Findings Summary

W4: Single-step authority transfer in MarginfiGroupConfigure

Impact:	Warning	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_group/configure.rs	Type:	Access control

Description

The `configure` instruction allows the current `admin` signer to update `marginfi_group.admin` immediately when `new_admin` is provided.

This transfer is single-step because the new authority does not need to acknowledge or accept the role in a separate instruction. As a result, a mistaken input or compromised admin signer can irreversibly move control to an unintended key in the same transaction.

Listing 5. Excerpt from configure

```
48 if let Some(new_admin) = new_admin {
49     marginfi_group.update_admin(new_admin);
50 }
```

Exploit scenario

1. Alice, the current protocol admin, submits the `configure` instruction with `new_admin` set to Bob's key.
2. The transaction executes and calls `update_admin` in the same instruction.
3. Bob, an unintended recipient or a compromised key holder, becomes admin immediately without an acceptance step.
4. Bob can exercise admin permissions right away, while Alice cannot cancel or confirm the handover in a dedicated second step.

Recommendation

Use a two-step admin transfer flow with a pending admin and explicit acceptance.

Acknowledgment 1.1

The client acknowledged the finding and provided the following comment:

> *It is done by multisig.*
>
> — Project0 team

Go back to Findings Summary

W5: New groups ignore global panic state until propagation

Impact:	Warning	Likelihood:	N/A
Target:		Type:	Logic error

Description

`initialize_group` copies fee parameters from global `fee_state` into `marginfi_group.fee_state_cache`, but it does not initialize `marginfi_group.panic_state_cache` from `fee_state.panic_state`.

Protocol pause checks use `group.is_protocol_paused()`, which reads only the group-local `panic_state_cache`. Therefore, if the global panic is active when a new group is created, the new group can still evaluate as unpaused.

Recommendation

Initialize each new group's `panic_state_cache` from global `fee_state.panic_state` during group creation.

Acknowledgment 1.1

The client acknowledged the finding and provided the following comment:

> *It is expected. We are running just one group.*
>
> — Project0 team

Go back to Findings Summary

W6: `global_fee_admin` and `global_fee_wallet` rotated in a single-step write with no two-step handover, timelock, or non-zero check

Impact:	Warning	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_group/edit_global_fee.rs	Type:	Access control

Description

`edit_fee_state` overwrites both `fee_state.global_fee_admin` and `fee_state.global_fee_wallet` in a single instruction, gated only by the `has_one = global_fee_admin` constraint on the incoming `FeeState` account. There is no two-step propose/accept flow, no timelock or cool-off period, and no sanity check that the new admin or wallet is non-zero, differs from `Pubkey::default()`, or differs from the current value. A single malformed transaction from a compromised or careless `global_fee_admin` is therefore sufficient to hand both the admin role and the protocol-fee destination to a different key.

Two distinct loss vectors follow from the missing guard rails:

- **Admin lock-out.** If `global_fee_admin` is rotated to a key nobody controls (for example `Pubkey::default()`, a typo, or an attacker's address), every `has_one = global_fee_admin`-gated instruction becomes permanently unreachable. That includes `panic_pause`, `panic_unpause`, `edit_fee_state` itself, `propagate_fee_state`, and the fee-admin path of `config_group_fee`. The protocol loses the ability to pause, unpause, adjust fees, or propagate fee-state changes to groups for the remaining lifetime of the deployment.
- **Wallet redirection.** If `global_fee_wallet` is rotated to an attacker-controlled address, every subsequent flat-SOL fee collected by `add_pool`,

`add_pool_with_seed`, `liquidate_end`, and `order`, together with every `collect_bank_fees` SPL payout, lands in the attacker's wallet until the admin notices and rotates back.

Because both fields are written in the same instruction, an operator error on one field (wrong admin pubkey) simultaneously disables the recovery path for the other (cannot call `edit_fee_state` again to fix the wallet).

Related: I8 covers the analogous issue on the initialization side, and L18 covers the adjacent gap that the same instruction also writes fee parameters without bounds checks.

Exploit scenario

Wallet-redirection scenario:

1. Bob, an attacker, controls a destination address. Alice, a compromised or careless `global_fee_admin`, calls `edit_fee_state` with `fee_wallet` set to Bob's address and the remaining parameters left at their current values.
2. The instruction overwrites `fee_state.global_fee_wallet` in a single write with no delay or confirmation.
3. Subsequent `collect_bank_fees`, `add_pool`, `add_pool_with_seed`, `liquidate_end`, and `order` flat-SOL and SPL protocol-fee payouts route to the attacker wallet until Alice notices and rotates back.

Admin-lock-out scenario:

1. Alice, a compromised or careless `global_fee_admin`, calls `edit_fee_state` with `admin` set to `Pubkey::default()` (or any key nobody controls).
2. The instruction overwrites `fee_state.global_fee_admin` in a single write with no delay or confirmation.
3. Every subsequent call to `panic_pause`, `panic_unpause`, `edit_fee_state`, `propagate_fee_state`, and the fee-admin path of `config_group_fee` fails the

`has_one = global_fee_admin` constraint — there is no on-chain recovery path.

Recommendation

Split the rotation into a two-step propose/accept flow: the current admin nominates a new admin (or wallet) into a pending slot, and the nominee must sign a separate `accept_fee_admin` / `accept_fee_wallet` instruction before the change takes effect. Reject `Pubkey::default()` and the unchanged-value case at write time, and add a mandatory timelock or on-chain signalled intent for the admin-role handover given the breadth of instructions that it gates. At minimum, separate the admin rotation and wallet rotation into distinct instructions so that an operator error on one field does not simultaneously hand over both protocol control and the fee destination, and so that either change can be reverted while the other is still live.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *it's done by multisig*
>
> — Project0 team

Go back to Findings Summary

I1: Permissionless bank creation event reports admin as signer

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_group/add_pool_permissionless.rs	Type:	Logging

Description

The `lending_pool_add_bank_permissionless` instruction is permissionless and is signed by `fee_payer`, so any user can create a bank when account constraints are satisfied.

However, the emitted `LendingPoolBankCreateEvent` sets `header.signer` to `group.admin` instead of the transaction signer. This makes the event attribute bank creation to the admin key even when the admin did not submit the instruction.

Listing 6. Excerpt from add_pool_permissionless

```
131 emit!(LendingPoolBankCreateEvent {
132     header: GroupEventHeader {
133         marginfi_group: ctx.accounts.marginfi_group.key(),
134         signer: Some(group.admin)
135     },
136     bank: bank_loader.key(),
137     mint: bank_mint.key(),
138 });
```

Recommendation

Emit the actual transaction signer in the event header for this permissionless instruction.

Fix 1.1

Fixed in commit `e5a3f83`[26] by updating the event emission to reflect the actual transaction signer.

Go back to Findings Summary

I2: `juplend_init_position` is permissionless and flips `operational_state` to `Operational`

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/juplend/init_position.rs	Type:	Access control

Description

Only `fee_payer: Signer` is required — anyone can pay the 10-unit seed and the CPI cost to activate a bank configured by the admin. Kamino's `init_obligation` is also permissionless but does not flip operational state. An attacker can activate a bank before the admin finalizes oracle or cap tuning (though `validate` ran at add-time), and can pin the seed deposit to the trivial minimum.

Recommendation

Gate behind `admin: Signer` matching `LendingPoolAddBankJuplend`, or require the signer to equal `group.admin`.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *we have process to initialize position when creating new bank*
>
> — Project0 team

Go back to Findings Summary

I3: `get_asset_shares` silently returns zero when `asset_share_value == 0`

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/state/bank.rs, programs/marginfi/src/state/marginfi_account.rs	Type:	Code quality

Description

`Bank::get_asset_shares` guards against division-by-zero by returning `Ok(I80F48::ZERO)` when `asset_share_value == 0`:

```
fn get_asset_shares(&self, value: I80F48) -> MarginfiResult<I80F48> {
    if self.asset_share_value == I80F48::ZERO.into() {
        return Ok(I80F48::ZERO);
    }
    ...
}
```

`socialize_loss` can set `asset_share_value = 0`; today the `handle_bankruptcy` path simultaneously transitions the bank to `KilledByBankruptcy` so deposits are structurally blocked. The two state changes are not structurally coupled in the accounting layer — any refactor that reaches this state without also killing the bank would cause deposit and repay to `transfer` tokens into the vault and credit **zero shares** to the user (silent fund loss).

Recommendation

Return `MarginfiError::BankAccountingError` (or a dedicated `BankKilledByBankruptcy` variant) when `asset_share_value == 0`, instead of silently returning zero shares. Mirror on the liability side.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *usually we like to batch multiple these IX, throwing an error will make complicated*
>
> — Project0 team

Go back to Findings Summary

I4: `lending_pool_withdraw_fees` has no mint check on `dst_token_account`

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_group/collect_bank_fees.rs	Type:	Data validation

Description

`dst_token_account: InterfaceAccount<TokenAccount>` is constrained only by `mut`. The SPL token program enforces the mint match at CPI time, but the pattern is inconsistent with `update_fees_destination_account`, which does validate mint explicitly.

The separate concern that `lending_pool_withdraw_fees` does not chain `accrue_interest` + `collect_bank_fees` before paying out — and so can forward stale vault balances — is covered in I16.

Recommendation

Add the mint constraint for consistency with `update_fees_destination_account`.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *token program handles check and it's OK*
>
> — Project0 team

Go back to Findings Summary

I5: `update_bank_cache` writes `last_update` independently of `accrue_interest`

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/state/bank.rs	Type:	Logic error

Description

`accrue_interest` sets `self.last_update = current_timestamp` passed in. `update_bank_cache` then unconditionally re-writes `self.last_update = Clock::get()?.unix_timestamp`. If any code path calls `update_bank_cache` standalone (without a preceding `accrue_interest`), interest is silently skipped for the interval `[previous_last_update, now]`.

Recommendation

Remove the `last_update` write from `update_bank_cache` — only `accrue_interest` must advance it. Alternatively, pass the timestamp into `update_bank_cache` as a parameter and avoid calling `Clock::get()` independently.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *We will keep the last_update write, indexers rely on it, not exploitable*
>
> — Project0 team

Go back to Findings Summary

I6: Several JupLend CPI ancillary accounts are fully unconstrained

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/juplend/deposit.rs, programs/marginfi/src/instructions/juplend/withdraw.rs, programs/marginfi/src/instructions/juplend/init_position.rs	Type:	Trust model

Description

Only `supply_token_reserves_liquidity` and `lending_supply_position_on_liquidity` are cross-checked against `integration_acc_1`. `rewards_rate_model` is present in the `Lending` struct but not enforced against it. `liquidity_program` is not pinned by address, unlike `juplend_program`. `rate_model`, `vault`, `liquidity`, and `lending_admin` are free-form.

Any defense-in-depth gap in the external program becomes a direct vulnerability through marginfi's authority-signed CPI.

Recommendation

At minimum, add `constraint = rewards_rate_model.key() == integration_acc_1.load()?.rewards_rate_model`. Pin `liquidity_program` by hardcoded address. Consider storing `vault`, `rate_model`, and `lending_admin` on `Bank` at `add_bank_juplend` time and enforcing via `has_one`.

Fix 1.1

The issue was fixed in commit `55ac554`[27]. The most security-critical ancillary accounts are now constrained: `liquidity_program` is pinned by hardcoded address to `JUPLEND_LIQUIDITY_PROGRAM_ID`, and `rewards_rate_model` is cross-

checked against `integration_acc_1.rewards_rate_model`.

Go back to Findings Summary

I7: `deposit_limit` on JupLend banks is silently denominated in fToken share units

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/state/juplend.rs, programs/marginfi/src/state/bank.rs	Type:	Logic error

Description

JupLend deposits credit `minted_shares` (fTokens) into `asset_shares`; `asset_share_value` stays at 1.0 (no interest accrual on the bank itself). The deposit-limit branch falls into the non-Drift `else` arm, so `deposit_limit` is interpreted in fToken units. Because `token_exchange_price >= 1e12` and only grows, the effective cap in underlying terms loosens over time, silently exceeding what the admin intended.

Recommendation

Add a JupLend branch in `get_remaining_deposit_capacity` / `change_asset_shares` (mirroring the Drift branch) that converts `deposit_limit` from underlying to fTokens using `token_exchange_price`, or rename the config field and document the unit explicitly.

Fix 1.1

The issue was fixed in commit `55ac554`[28]. Doc comments were added on the JupLend, Kamino, and Solend `ConfigCompact.deposit_limit` fields clarifying that the limit is denominated in collateral / fToken units rather than in the underlying asset. The client added the following comment.

> *comment was added on how to interpret deposit limit*

> — Project0 team

Go back to Findings Summary

I8: `init_global_fee_state` has no authority check on the payer

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_group/init_global_fee_state.rs	Type:	Access control

Description

`InitFeeState` requires only an unconstrained `payer: Signer`, and the instruction accepts `admin_key` and `fee_wallet` as caller-chosen parameters that are written directly into `FeeState`:

```
#[derive(Accounts)]
pub struct InitFeeState<'info> {
    /// Pays the init fee
    #[account(mut)]
    pub payer: Signer<'info>,

    #[account(
        init,
        seeds = [
            FEE_STATE_SEED.as_bytes()
        ],
        bump,
        payer = payer,
        space = 8 + FeeState::LEN,
    )]
    pub fee_state: AccountLoader<'info, FeeState>,

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

```
) -> Result<()> {
    let mut fee_state = ctx.accounts.fee_state.load_init()?;
    fee_state.global_fee_admin = admin_key;
    fee_state.global_fee_wallet = fee_wallet;
    fee_state.key = ctx.accounts.fee_state.key();
```

```
        fee_state.bank_init_flat_sol_fee = bank_init_flat_sol_fee;
        fee_state.bump_seed = ctx.bumps.fee_state;
        fee_state.program_fee_fixed = program_fee_fixed;
        fee_state.program_fee_rate = program_fee_rate;
        fee_state.liquidation_max_fee = liquidation_max_fee;
        fee_state.liquidation_flat_sol_fee = liquidation_flat_sol_fee;
        fee_state.order_execution_max_fee = order_execution_max_fee;
        fee_state.order_init_flat_sol_fee = order_init_flat_sol_fee;

        Ok(())
    }
```

On the current mainnet deployment this is unexploitable — the `FeeState` PDA is already initialized and Anchor's `init` constraint prevents re-initialization. However, the program carries no compile-time protection against a future re-deploy scenario (new cluster, fork, devnet refresh, or test environment): the first caller wins the race and permanently controls `edit_global_fee_state`, `config_group_fee`, and `panic_pause` via the `global_fee_admin` they installed for themselves.

Recommendation

Constrain `payer` with `address = crate::EXPECTED_DEPLOYER` against a hardcoded deployer pubkey, or derive the admin from the program's BPF upgrade authority at init time instead of accepting `admin_key` as an instruction parameter. Either approach eliminates the first-caller-wins race on any future re-deploy.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *per-program one-time-only instruction, no meaningful payer constraints*
>
> — Project0 team

Go back to Findings Summary

I9: JupLend fToken vault seed constant mismatch between type-crate and program

Impact:	Info	Likelihood:	N/A
Target:	type-crate/src/constants.rs, programs/marginfi/src/constants.rs	Type:	Code quality

Description

The fToken vault PDA seed is defined differently in two locations:

- `type-crate/src/constants.rs`: `JUPLEND_F_TOKEN_VAULT_SEED = "juplend_f_token_vault"`
- `programs/marginfi/src/constants.rs`: `JUPLEND_F_TOKEN_VAULT_SEED = "f_token_vault"`

The on-chain program uses `"f_token_vault"`, which is the correct and authoritative value. However, any external client, SDK, or indexer that imports the type-crate constant will derive an incorrect PDA address.

There is no on-chain impact — the program uses the correct seed internally. Off-chain clients using the type-crate constant will fail to derive valid account addresses for JupLend fToken vaults, causing transaction-construction failures.

Recommendation

Update the `JUPLEND_F_TOKEN_VAULT_SEED` constant in the type-crate to the string `"f_token_vault"` so it matches the authoritative value used by the on-chain program.

Fix 1.1

The issue was fixed in commit `9a177a7`[29]. The `JUPLEND_F_TOKEN_VAULT_SEED` constant in the type-crate was updated to `"f_token_vault"` to match the authoritative value used by the on-chain program, so external clients now derive the correct PDA.

Go back to Findings Summary

I10: `MarginfiAccount` init and close instructions bypass the protocol pause check

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_account/initialize.rs, programs/marginfi/src/instructions/marginfi_account/close.rs, programs/marginfi/src/state/marginfi_account.rs	Type:	Access control

Description

Nearly every user-facing instruction in the protocol gates its accounts with `constraint = !group.load()?.is_protocol_paused() @ MarginfiError::ProtocolPaused`. Two `MarginfiAccount` lifecycle instructions break that invariant:

- **Initialize** — `MarginfiAccountInitialize` and `MarginfiAccountInitializePda` include the `MarginfiGroup` account but do not apply the `is_protocol_paused()` constraint. New accounts can therefore be created during a panic pause. There is no direct exploit path today — creating an empty account during a pause has no immediate consequence because all subsequent operations (deposit, borrow, etc.) enforce their own pause checks — but it is inconsistent with the protocol's pause design and could mask unexpected state transitions if combined with future changes.
- **Close** — `MarginfiAccountClose` takes no `MarginfiGroup` account at all, so `is_protocol_paused()` cannot be evaluated at account-struct level. Separately, `can_be_closed` does not reject `ACCOUNT_IN_ORDER_EXECUTION` or `ACCOUNT_IN_DELEVERAGE`; today these imply non-empty balances so the

`only_has_empty_balances` guard catches them, but the coupling is implicit and fragile.

The two cases share the same class (account-lifecycle instructions skip the pause gate) and warrant the same fix shape.

Recommendation

Add the `is_protocol_paused()` constraint to both `MarginfiAccountInitialize` and `MarginfiAccountInitializePda`. Add `group: AccountLoader<'info, MarginfiGroup>` to `MarginfiAccountClose` with `has_one` plus the pause constraint, and extend `can_be_closed` to explicitly reject all session flags (order execution, deleverage, receivership) rather than relying on the emptiness guard.

Fix 1.1

The issue was fixed in commit `55ac554`[30]. A `!is_protocol_paused()` constraint was added to both `MarginfiAccountInitialize` and `MarginfiAccountClose`, so account initialization and closure now respect the global protocol pause.

Go back to Findings Summary

I11: `marginfi_account_init_liq_record` has no authority/`has_one`/pause check

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_account/init_liquid_record.rs	Type:	Access control

Description

The instruction writes `marginfi_account.liquidation_record = liquidation_record.key()` without a signer check, a `has_one = group`, a pause constraint, or a `check_eq!(marginfi_account.liquidation_record, Pubkey::default())` preamble. Today the `init` PDA collision blocks creating a second record, but the field overwrite itself is not defended.

Recommendation

Require `marginfi_account.liquidation_record == Pubkey::default()` before writing; add a pause constraint.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *anyone should be able to create that account, purely for off-chain tracking*
>
> — Project0 team

Go back to Findings Summary

I12: `marginfi_account_initialize_pda` — `Some(0)` and `None third_party_id` derive the same PDA

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_account/initialize.rs, programs/marginfi/src/constants.rs	Type:	Code quality

Description

The PDA seeds use `third_party_id.unwrap_or(0)`, so `None` and `Some(0)` produce the same address. The CPI-allowlist check runs only in the `Some(id)` branch. Today `is_allowed_cpi_for_third_party_id` short-circuits to `Ok(true)` for `id < PDA_FREE_THRESHOLD`, so the two branches are functionally identical, but the structural redundancy invites future bugs if the semantics change.

Recommendation

Either always run the check (`third_party_id.unwrap_or(0)`), or reject `Some(0)` explicitly and require `None` for the free tier.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *that's not a problem*
>
> — Project0 team

Go back to Findings Summary

I13: `place_order` reuses existing balance tags; `set_keeper_close_flags` has per-balance scope

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_account/order.rs	Type:	Code quality

Description

Two orders on overlapping bank pairs `(A, B)` and `(A, C)` share the same tag on balance `A`. `set_keeper_close_flags` clears balance `A's tag, which simultaneously makes both orders keeper-closeable. Combined with the fact that `set_keeper_close_flags` bypasses the account freeze, this means authority-driven flag-clearing cascades across orders. Not exploitable today; fragile for future per-order keeper actions.

Recommendation

Either document the shared-tag semantics explicitly, or migrate to per-order unique tags stored on `Order`.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *it's going to be documented*
>
> — Project0 team

Go back to Findings Summary

I14: `keeper_close_order` event carries `Pubkey::default()` when `marginfi_account` is already closed

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_account/order.rs	Type:	Logging

Description

In the "marginfi account already closed" branch, the emitted `KeeperCloseOrderEvent` reports `marginfi_account_authority = Pubkey::default()` and `marginfi_group = Pubkey::default()`, defeating off-chain indexers keying on these fields.

Recommendation

Store `authority` and `group` on `Order` at init time and use them here.

Fix 1.1

The issue was fixed in commit `a5fda8c`[31]. The fall-through deserialization branch in `keeper_close_order` is unchanged, but `marginfi_account_close` now enforces `active_orders == 0` before closing, so the "marginfi account already closed but order PDA still open" branch where the event carried `Pubkey::default()` is no longer reachable. The client added the following comment.

> *currently user cannot close account if they have any order*
>
> — Project0 team

Go back to Findings Summary

I15: Order-execution `validate_ix_first` rejects allowlisted ATA `Create` / `CreateIdempotent`

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_account/liquidate_start.rs, programs/marginfi/src/ix_utils.rs	Type:	Logic error

Description

`validate_instructions` allowlists the Associated Token Account program, but `validate_ix_first` rejects any non-compute instruction whose `data.len() < 8` with `StartNotFirst`. ATA `Create` / `CreateIdempotent` use a 1-byte payload — so the ATA program is listed but unusable inside order-execution transactions.

This is operational only: keepers must use a pre-existing destination token account (which is how current tests drive orders anyway). No funds are at risk and no invariant is bypassed.

Recommendation

Either drop `ASSOCIATED_TOKEN_KEY` from the allowlist (clearer intent), or special-case ATA-program instructions in `validate_ix_first` to skip the 8-byte length gate.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *we like to whitelist as small number of ix and programs as possible*

> — Project0 team

Go back to Findings Summary

I16: Several handlers read and act on bank state without first calling `accrue_interest`

Impact:	Info	Likelihood:	N/A
Target:	programs/marginfi/src/instructions/marginfi_account/emissions.rs, programs/marginfi/src/state/marginfi_account.rs, programs/marginfi/src/instructions/marginfi_group/collect_bank_fees.rs	Type:	Logic error

Description

Several handlers read bank state that is a function of `accrue_interest` — share values and `collected_*_outstanding` — and transfer or pay out based on that state without first advancing accrual. In low-activity banks the driving state is whatever the last deposit, withdraw, borrow, or repay wrote, which can be hours old.

Emissions handlers (`claim_emissions` and `settle_emissions_and_get_transfer_amount`) scale by `bank.get_asset_amount(balance.asset_shares)` or `get_liability_amount(…)`, which use the bank's current share value. None of the four emissions entry points calls `bank.accrue_interest` before running the conversion, so the payout uses stale shares. The direction is systematically against the user (under-payment), and the user cannot recover the gap because `claim_emissions` always advances `balance.last_update` to `now`.

`lending_pool_collect_bank_fees` snapshots `collected_*_outstanding` and moves those tokens to the fee vault without first calling `accrue_interest`, so

any fees accrued since the last `accrue_interest` call are not included and the fee vault lags real accrual.

`lending_pool_withdraw_fees` compounds this by paying `dst_token_account` out of the current fee vault balance without chaining either `accrue_interest` or `collect_bank_fees` first, so a crank that expects "withdraw → up-to-date payout" can forward a stale balance unless the operator manually sequences `accrue_interest` + `collect_bank_fees` + `withdraw_fees` in the same transaction.

Recommendation

Call `bank.accrue_interest(Clock::get()?.unix_timestamp, &group, …)` at the top of each emissions handler and at the top of `lending_pool_collect_bank_fees`. In `lending_pool_withdraw_fees`, chain `accrue_interest` followed by `collect_bank_fees` before the vault transfer, or document the required crank sequence. Add `group` to `LendingAccountSettleEmissions` as part of the H4 fix.

Fix 1.1

The issue was fixed in commit `f28865d`[32]. The emissions handlers cited in the description were removed as part of the emissions wind-down, so they no longer read share values without first calling `accrue_interest`. The client kept the `lending_pool_collect_bank_fees` and `lending_pool_withdraw_fees` accrual behavior unchanged. The client added the following comment.

> *emission path is fixed by removing emissions, other are not applicable*
>
> — Project0 team

Go back to Findings Summary

[1] full commit hash: `2e5953f8c66732be022985a063cd582ad481f1a3`, link to commit

[2] full commit hash: `390dae2366e0d6e05797cfef34db6f21dda8754a`, link to commit

[3] full commit hash: `8f38cfb9109cbc7ee78cea6fe4c4e9a925933122`, link to commit

[4] full commit hash: `9e8a881eac153fbc50ccf6409d7c1107bcdb7832`, link to commit

[5] full commit hash: `f28865dc05439e9f41a78c3a61cbcc513a06c3f8`, link to commit

[6] full commit hash: `ef7768874ae362a0a20cd6737781941a5b6cda6b`, link to commit

[7] full commit hash: `4a20de1fb5af9afcc4ec6c35f71754298adc612e`, link to commit

[8] full commit hash: `8f38cfb9109cbc7ee78cea6fe4c4e9a925933122`, link to commit

[9] full commit hash: `8f38cfb9109cbc7ee78cea6fe4c4e9a925933122`, link to commit

[10] full commit hash: `f28865dc05439e9f41a78c3a61cbcc513a06c3f8`, link to commit

[11] full commit hash: `186b248d112bbd4dd405acc0f033cd041c1c843d`, link to commit

[12] full commit hash: `f28865dc05439e9f41a78c3a61cbcc513a06c3f8`, link to commit

[13] full commit hash: `9e8a881eac153fbc50ccf6409d7c1107bcdb7832`, link to commit

[14] full commit hash: `f28865dc05439e9f41a78c3a61cbcc513a06c3f8`, link to commit

[15] full commit hash: `f28865dc05439e9f41a78c3a61cbcc513a06c3f8`, link to commit

[16] full commit hash: `186b248d112bbd4dd405acc0f033cd041c1c843d`, link to commit

[17] full commit hash: `390dae2366e0d6e05797cfef34db6f21dda8754a`, link to commit

[18] full commit hash: `984cefa158adffa6aab29052041f020e8f6bd652`, link to commit

[19] full commit hash: `f28865dc05439e9f41a78c3a61cbcc513a06c3f8`, link to commit

[20] full commit hash: `f28865dc05439e9f41a78c3a61cbcc513a06c3f8`, link to commit

[21] full commit hash: `f28865dc05439e9f41a78c3a61cbcc513a06c3f8`, link to commit

[22] full commit hash: `f28865dc05439e9f41a78c3a61cbcc513a06c3f8`, link to commit

[23] full commit hash: `a5fda8c1bf5861791763b81650b86b4c55109eef`, link to commit

[24] full commit hash: `55ac5544669bc0cc422e2c20a7441b4ade2dafd5`, link to commit

[25] full commit hash: `5b8337ddc6e34c0046fdd8540a17162d7024a6ee`, link to commit

[26] full commit hash: `e5a3f830c84eca1218512cf700376225948917f8`, link to commit

[27] full commit hash: `55ac5544669bc0cc422e2c20a7441b4ade2dafd5`, link to commit

[28] full commit hash: `55ac5544669bc0cc422e2c20a7441b4ade2dafd5`, link to commit

[29] full commit hash: `9a177a771ae90949950b12d2bce15c53b7e664f4`, link to commit

[30] full commit hash: `55ac5544669bc0cc422e2c20a7441b4ade2dafd5`, link to commit

[31] full commit hash: `a5fda8c1bf5861791763b81650b86b4c55109eef`, link to commit

[32] full commit hash: `f28865dc05439e9f41a78c3a61cbcc513a06c3f8`, link to commit

Report Revision 1.1

Revision Team

Member's Name	Position
Andrej Lukačovič	Lead Auditor
Max Kupchenko	Auditor
Matej Hyčko	Auditor
Artem Delikatnyi	Auditor
Yulian Kisil	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

Overview

Since there were no comprehensive changes in this revision, the complete overview is listed in the Executive Summary section Revision 1.1.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Audit Report | Project0: DeFi-Native Prime Broker, June 3, 2026.

Appendix B: Trident Findings

This section details the results and methodology of fuzz testing performed using the Trident framework during the audit. The audit included Trident-based fuzzing in `trident-tests/` targeting the deployed `marginfi` program interface and its risk-engine/oracle integration patterns.

B.1. Fuzzing

Mainnet State Forking (real dependencies): The fuzz harness uses Trident fork configuration (`trident-tests/Trident.toml`) to fetch and cache external on-chain accounts (e.g., Pyth Push oracle `PriceUpdateV2` accounts, Kamino program accounts) from mainnet. This keeps oracle and integration-account layouts realistic rather than mocked.

Oracle Mutation (price shocks): To drive liquidations and health transitions, `fuzz_0` mutates forked Pyth Push oracle accounts in-place (`trident-tests/fuzz_0/oracle_patch.rs`) by scaling spot and EMA price/confidence fields and refreshing publish timestamps to match the fuzzed clock.

Time Manipulation + Interest Accrual: `fuzz_0` advances time between actions using `forward_in_time()` and then permissionlessly accrues interest across all fuzz banks (`lending_pool_accrue_bank_interest`) to ensure `Bank.last_update` evolves and interest paths are exercised.

Token-2022 Account-Tail Coverage: The suite includes Token-2022 mints and verifies correct remaining-account ordering (e.g., prepending the bank mint only when required) to cover Token-2022 transfer paths used by deposits/withdraws/borrows/repays.

Liquidation + Receivership Sessions: `fuzz_0` includes both direct liquidations and receivership flows ("start_liquidation → ... → end_liquidation") and checks that session flags clear correctly at the end of successful bundles.

Integration Coverage (Kamino + Juplend): `fuzz_0` exercises Kamino and Juplend integration instructions (USDC) and asserts balance/share invariants across the external programs' reserve/vault accounts.

The following table lists all implemented execution flows in the Trident fuzz testing framework.

ID	Flow	Added
F1	fuzz_0: Initialize FeeState, create a MarginfiGroup, add a bank, configure oracle, initialize a MarginfiAccount, and init its LiquidationRecord PDA	1.0
F2	fuzz_0: Seed initial liquidity by depositing into USDC/ETH/BTC banks	1.0
F3	fuzz_0: Random user deposits USDC with randomized amount	1.0
F4	fuzz_0: Random user withdraws USDC with randomized amount	1.0
F5	fuzz_0: Random user borrows ETH with randomized amount	1.0
F6	fuzz_0: Random user repays ETH with randomized amount	1.0
F7	fuzz_0: Random user executes a BTC flashloan (borrow vs repay randomized; matched amounts should succeed, mismatched should revert)	1.0
F8	fuzz_0: Liquidate a random liquidatee (USDC asset vs ETH liability) under randomized oracle price scaling	1.0
F9	fuzz_0: Receivership liquidation bundle (start/end liquidation session) under randomized oracle price scaling	1.0

ID	Flow	Added
F10	fuzz_0: Deposit to Kamino obligation (USDC) and withdraw from obligation (USDC)	1.0
F11	fuzz_0: Deposit to Juplend position (USDC) and withdraw from position (USDC)	1.0
F12	fuzz_0: Forward time by a randomized delta and accrue interest across all banks (and refresh Pyth publish_time)	1.0

Table 4. Trident fuzzing flows

The following table lists all implemented invariant checks in the Trident fuzz testing framework.

ID	Invariant	Added	Status
IV1	Deposit/repay token legs never increase user token balances or decrease the bank liquidity vault balance	1.0	Success
IV2	Withdraw/borrow token legs never decrease user token balances or increase the bank liquidity vault balance	1.0	Success
IV3	User token delta and bank liquidity-vault delta are conserved per successful deposit/withdraw/borrow/repay	1.0	Success
IV4	Successful deposits increase asset shares and do not change liability shares for that bank	1.0	Success
IV5	Successful withdraws decrease asset shares (or close the position) and do not change liability shares for that bank	1.0	Success

ID	Invariant	Added	Status
IV6	Successful borrows increase liability shares and do not change asset shares for that bank	1.0	Success
IV7	Successful repays decrease liability shares (or close the position) and do not change asset shares for that bank	1.0	Success
IV8	Operations touching one bank do not change unrelated banks' liquidity vault balances	1.0	Success
IV9	Flashloan closed loop: when borrow_amount == repay_amount and the transaction succeeds, the user token balance is unchanged net	1.0	Success
IV10	Flashloan mismatch: when borrow_amount != repay_amount, the flashloan transaction must revert	1.0	Success
IV11	Liquidation conserves total liability tokens across liability liquidity-vault + insurance-vault, and moves insurance fees in the correct direction	1.0	Success
IV12	Successful liquidation decreases liquidatee USDC asset shares and ETH liability shares, and increases the liquidator's corresponding shares	1.0	Success
IV13	Failed liquidation does not change vault balances or the tracked marginfi share snapshots	1.0	Success

ID	Invariant	Added	Status
IV14	Receivership: after a successful start→…→end bundle, ACCOUNT_IN_RECEIVERSHIP (and other session flags) are cleared and the LiquidationRecord receiver is reset	1.0	Success
IV15	Kamino deposit (success): user outflow matches reserve supply inflow; marginfi vault is net unchanged; obligation collateral destination increases (or no movement for amount == 0)	1.0	Success
IV16	Kamino deposit (failure): user / marginfi vault / reserve supply / collateral destination balances remain unchanged	1.0	Success
IV17	Kamino withdraw (success): user increases consistently with reserve supply decrease; marginfi vault is net unchanged; obligation collateral source decreases (handles withdraw_all)	1.0	Success
IV18	Kamino withdraw (failure): user / marginfi vault / reserve supply / collateral source balances remain unchanged	1.0	Success
IV19	Juplend deposit (success): user decreases; fToken vault increases; marginfi liquidity vault is net unchanged; amount == 0 is no movement	1.0	Success
IV20	Juplend deposit (failure): user / marginfi vault / fToken vault remain unchanged	1.0	Success

ID	Invariant	Added	Status
IV21	Juplend withdraw (success): user increases; fToken vault decreases; withdraw intermediary ATA and marginfi liquidity vault are net unchanged; withdraw_all handled	1.0	Success
IV22	Juplend withdraw (failure): user / withdraw intermediary ATA / marginfi vault / fToken vault remain unchanged	1.0	Success
IV23	After forward_in_time() + successful accrue, each bank's last_update must strictly advance	1.0	Success

Table 5. Trident fuzzing invariants

Appendix C: Trident Arena Findings

This section lists vulnerabilities identified by Trident Arena, an automated vulnerability analysis system designed for Solana program reviews. Trident Arena uses multi-agent AI as one of its core analysis engines and layers additional checks on top, complementing manual review.

C.1. Discovered Findings

The following table contains true-positive findings identified by Trident Arena. These findings are included regardless of whether they were also discovered independently by auditors during manual review.

Finding title	Severity	Reported	Discoverer
H1: Receivership liquidation bypass via dangling LiquidationRecord linkage	**High**	1.0	Trident Arena, Auditor
M3: Bank creation allows unbounded Token-2022 mint extensions	Medium	1.0	Trident Arena, Auditor
M4: Permissionless propagation can override manual staked bank configuration	Medium	1.0	Trident Arena, Auditor
L3: Emode clone skips configuration validation	Low	1.0	Trident Arena, Auditor
L4: Isolated assets can be stranded after equity-based bankruptcy disablement	Low	1.0	Trident Arena, Auditor

Finding title	Severity	Reported	Discoverer
W4: Single-step authority transfer in MarginfiGroupConfigure	Warning	1.0	Trident Arena, Auditor
W5: New groups ignore global panic state until propagation	Warning	1.0	Trident Arena, Auditor
I1: Permissionless bank creation event reports admin as signer	Info	1.0	Trident Arena, Auditor

Table 6. Table of findings identified by Trident Arena

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz