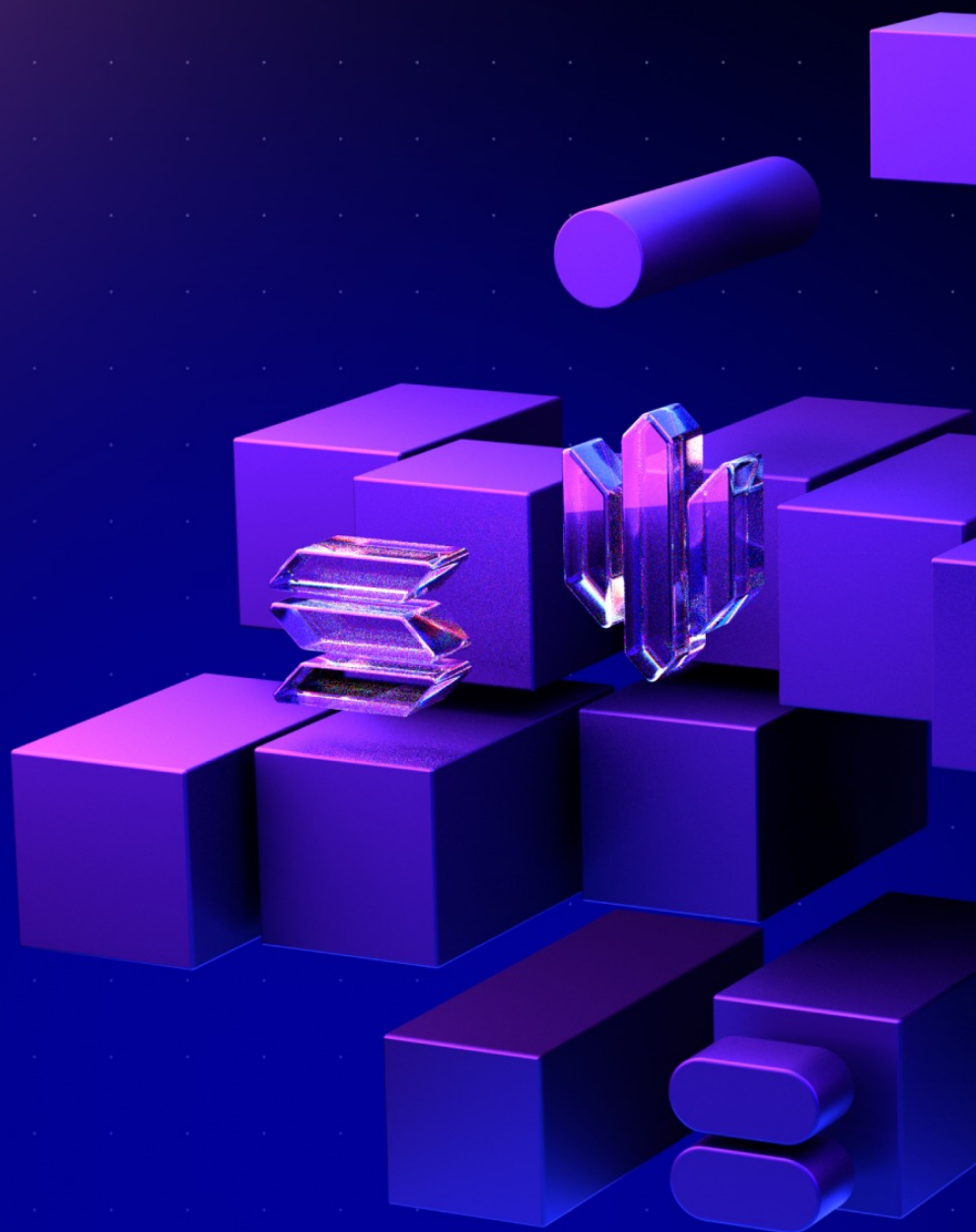
ackee
blockchain security

Hobba

Prime broker for onchain loans on Solana

June 15, 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
   Revision 1.2 . . . 13
4. Findings Summary . . . 15
Report Revision 1.0 . . . 20
   Revision Team . . . 20
   System Overview . . . 20
   Trust Model . . . 21
   Fuzzing . . . 22
   Findings . . . 23
Report Revision 1.1 . . . 91
   Revision Team . . . 91
   Findings . . . 91
Report Revision 1.2 . . . 100
   Revision Team . . . 100
Appendix A: How to cite . . . 101
Appendix B: Trident Arena Findings . . . 102
   B.1. Discovered Findings . . . 102

1. Document Revisions

1.0	Final Report	May 26, 2026
1.1	Fix Review	June 10, 2026
1.2	Fix Review	June 15, 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
Matej Hyčko	Auditor
Artem Delikatnyi	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

Hobba is a Solana-based leveraged yield protocol that lets users deposit collateral (e.g., cbBTC) and borrow USDC against it, then deploys the borrowed liquidity into supported yield vaults to earn yield while managing the position's target LTV. It uses on-chain state (PDAs) and CPI integrations with external lending markets (Kamino KLend and JupLend) and vault providers (Kamino KVault and Perena), with an operator role that can rebalance, repay, harvest profits, and migrate positions between providers.

Revision 1.0

Hobba engaged Ackee Blockchain Security to perform a security review of Hobba with a total time donation of 13 engineering days in a period between May 4 and May 26, 2026, with Andrej Lukačovič as the lead auditor.

The audit was performed on the commit `62f5f0e`[1] in the Hobba repository.

The audit was split into multiple phases. In the first phase, we focused on understanding the whole scope and the intended behavior of the system, with emphasis on how Hobba orchestrates user positions across external protocols via CPI and how operator-managed flows are expected to be used. This phase was dedicated to mapping key assets and fund flows (collateral and USDC), identifying trust boundaries (admin/operator, PDA custody), and establishing the expected invariants for deposit/borrow/repay/withdraw and vault routing.

The second phase was dedicated to an in-depth analysis of the code and its integrations (e.g., Kamino KLend, JupLend, and vault providers). During the manual review we paid special attention to the following:

- ensuring the math and accounting operations are correctly implemented;
- ensuring all of the accounts are correctly constrained and no misuse is

possible (especially `UncheckedAccount` and `remaining_accounts`-driven CPIs);

- ensuring no one can benefit from truncating, rounding, or overflow/underflow edge cases; and
- ensuring the cross-program communication is correctly implemented and cannot be abused to desynchronize Hobba state from external provider state.

The third phase of the audit focused on consolidating our findings, developing proof-of-concept demonstrations where necessary to show impact, and compiling the audit report.

As the final phase of the audit, we employed Trident Arena, a multi-agent AI system, which provided additional analysis and cross-checking of the identified issues. See the Appendix B for the complete list of findings identified by Trident Arena.

Our review resulted in 24 findings, ranging from Info to High severity.

The H1, where the migration unwind/windup instructions lack on-chain phase gating; an operator can execute the legs out of order (or as standalone calls), desynchronize `UserState` from the actual provider position, and extract unearned migration fees by self-funding the unwind and later reclaiming `flash_amount + fee` on windup.

The M2, where user-facing destination token accounts are not bound to canonical ATAs; if a user signs a transaction built by a malicious/compromised frontend, assets can be transferred to an attacker-controlled account.

The M1, where `claim_vault_rewards` accepts reward destinations via `remaining_accounts` and does not validate the Associated Token Program; an operator can redirect farm rewards to operator-controlled token accounts.

The M3, where the JupLend borrow path can skip borrowing the remainder

after a partial vault withdrawal, causing the instruction to succeed while transferring less USDC than requested.

The M4, where `claim_vault_rewards` requests `user_state` as writable in an inner CPI while the outer account is read-only, leading to a privilege escalation failure and blocked reward claiming.

The L8, where `switch_vault` does not require the old share source account to be the canonical ATA; an operator can switch the user's configured vault while leaving old vault shares stranded in the canonical account.

The L9, where Perena vault dispatch builds CPIs from raw `vault_remaining` accounts without binding the vault state and mints to the named accounts; an operator can cause Hobba's accounting to reference one vault while the CPI operates on another.

The L7, where JupLend `operate` CPIs do not bind the remaining account graph to `UserState`'s recorded position/config; this can lead to untracked debt/collateral and state/CPI desynchronization.

The L2, where `claim_vault_rewards` does not constrain the Associated Token Program before invoking ATA-creation CPI, allowing an arbitrary program to be invoked in its place.

The L3, where `set_treasury` can misconfigure a treasury address in a way that blocks `harvest_vault` fee transfers.

The L4, where `migrate_juplend_unwind` does not verify collateral is fully withdrawn, potentially leaving residual collateral in the source position.

The L5, where `deposit` does not bind `supply_token_mint` to the `UserState` collateral mint, making the deposit path easier to misuse with mismatched mints/accounts.

The L6, where Perena vault flows inconsistently enforce min-out/slippage

protections and sometimes pass `0`, disabling on-chain bounds.

The L1, where Kamino and JupLend position accounting is read without required refresh, causing stale repay sizing and incorrect health assumptions.

Overall, the findings indicate that Hobba's current codebase and integration patterns have significant room for improvement; the code quality and consistency should be improved before going into production.

Ackee Blockchain Security recommends Hobba:

- improve account validation and binding across all entrypoints (especially `UncheckedAccount`, `remaining_accounts`, and external program id constraints);
- reduce bespoke CPI/account-parsing logic where possible by reusing well-audited integration patterns;
- improve overall code quality and consistency before any production deployment;
- 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

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

The review was performed between June 2 and June 10, 2026 on the commit `685a17a`[2].

From the reported 24 findings:

- 18 issues were fixed;
- 5 issues were acknowledged;
- 1 issue was fixed partially.

Two new issues were reported.

The H2 insufficient validation of the Kamino program id, enabling arbitrary CPI with UserState PDA signer.

The W8 operator JupLend harvest does not pin the JupLend program id for `update_exchange_prices` CPI.

Overall, the fixes materially improved account validation, CPI constraints, and code consistency across the previously reported findings; however, the newly reported H2 indicates that further hardening is still required.

Ackee Blockchain Security recommends Hobba:

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

Revision 1.2

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

The review was performed between June 14 and June 14, 2026 on the commit `dbfa246`[3].

From the reported 2 findings:

- 2 issues were fixed.

No new issues were reported.

[1] full commit hash: `62f5f0e35edfc2d7d281cb19036b857a93daab0f`

[2] full commit hash: `685a17a5f545502f6d960d141bffae5500fa4ca6`

[3] full commit hash: `dbfa24628b5010c4609c6ce06f572389106d9e01`

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	2	4	9	8	3	**26**

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
H1: Migration windup/unwind lack on-chain phase gating, enabling standalone operator-funded migrations and unearned fee extraction	**High**	1.0	Fixed
M1: claim_vault_rewards does not bind reward destinations nor validate the Associated Token Program, enabling reward redirection and arbitrary CPI invocation	Medium	1.0	Fixed

Finding title	Severity	Reported	Status
M2: User-facing token accounts are not bound to canonical ATAs, enabling destination substitution in user-signed flows	Medium	1.0	Fixed
M3: JupLend borrow skips lender borrow after partial vault withdrawal	Medium	1.0	Acknowledged
M4: claim_vault_rewards passes UserState as a writable CPI signer while the outer account is read-only	Medium	1.0	Fixed
L1: Kamino and JupLend position accounting is read without required refresh, causing stale repay sizing and incorrect health assumptions	Low	1.0	Acknowledged
L2: claim_vault_rewards does not validate the Associated Token Program before invoking ATA creation CPI	Low	1.0	Fixed
L3: set_treasury allows misconfiguration that can block harvest_vault fee transfers	Low	1.0	Fixed
L4: migrate_juplend_unwind does not verify collateral is fully withdrawn	Low	1.0	Fixed

Finding title	Severity	Reported	Status
L5: deposit does not bind supply_token_mint to user_state collateral mint	Low	1.0	Fixed
L6: Vault flows have inconsistent min-out / slippage enforcement (Perena)	Low	1.0	Fixed
L7: JupLend flows do not bind operate account list to UserState fields, allowing state/CPI desynchronization	Low	1.0	Fixed
L8: switch_vault accepts a non-canonical source share account and can strand old vault shares	Low	1.0	Fixed
L9: Perena vault CPIs do not bind vault_remaining to the named vault and mints	Low	1.0	Fixed
W1: Hardcoded third-party account parsing lacks account-type validation, making integrations fragile to wrong accounts and upstream layout changes	Warning	1.0	Partially fixed
W2: External account readers truncate u128 values to u64 via `as u64`, which can silently wrap at 2^64	Warning	1.0	Fixed

Finding title	Severity	Reported	Status
W3: Kamino collateral reader sums all deposit entries but converts using a single reserve exchange rate	Warning	1.0	Fixed
W4: JupLend prepare_withdraw computes repay sizing from vault_state without refreshing exchange prices	Warning	1.0	Fixed
W5: Migration windup trusts user-supplied Kamino flash fee when computing new borrow amount	Warning	1.0	Fixed
W6: initialize_program is unrestricted and allows the first caller to become admin	Warning	1.0	Fixed
W7: Allowlist enforcement applies to exit actions and can block user repayments and withdrawals	Warning	1.0	Acknowledged
I1: Supported collateral and debt assets are not enforced on-chain	Info	1.0	Acknowledged
I2: Several instructions include unused System Program accounts	Info	1.0	Fixed
I3: deposit does not validate token_program and uses it as CPI target	Info	1.0	Fixed

Finding title	Severity	Reported	Status
H2: Kamino CPIs do not validate the Kamino program id, enabling arbitrary CPI with UserState PDA signer	**High**	1.1	Fixed
W8: Operator JupLend harvest does not pin the JupLend program id for update_exchange_prices CPI	Warning	1.1	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

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

System Overview

Hobba is a Solana program that manages user positions across multiple integrated DeFi providers. It uses a per-user Program Derived Address (PDA) (`UserState`) as a custody and signing authority for protocol interactions, and routes deposits, borrows, repays, withdrawals, and "vault" operations through CPI into external programs.

UserState (PDA) and configuration. Each user position is represented by a `UserState` PDA keyed by `(owner, collateral_mint)` and stores the current lending provider selection plus the tracked external accounts (e.g., obligation/position and lending market). Global configuration lives in `ProgramConfig`, with per-collateral configuration in `DepositConfig` and optional allowlist gating.

Lending providers. Hobba integrates with **Kamino KLend** and **JupLend** as alternative lending backends. Depending on `user_state.lending_provider`, user flows execute provider-specific CPIs to deposit collateral, borrow USDC, repay debt, and withdraw collateral.

Vault providers. In addition to direct lending, Hobba routes USDC through vault providers (e.g., **KVault** and **Perena**) via a shared dispatcher. Vault shares

are held by the `UserState` PDA and can be withdrawn back to USDC to satisfy borrows/repays or harvested for fees.

Operator-managed flows. Some actions are reserved for an assigned **operator** (recorded in `UserState` and/or `ProgramConfig`), including reward claiming, vault switching/harvesting, and cross-provider migrations. Migration flows are multi-step sequences intended to be executed atomically (flash borrow → unwind → windup → flash repay), and rely on correct ordering and parameterization to preserve user safety.

Trust Model

Hobba program admin and configuration

Hobba includes privileged configuration and operational roles (e.g., admin/upgrade authority and operator).

Users must trust that:

- the **admin / upgrade authority** will not misconfigure critical parameters (e.g., allowlist policy, treasury, fee settings) in ways that can block user exits or redirect value; and
- protocol upgrades will not introduce logic that changes custody/authorization assumptions for existing user positions.

Operator role

Certain flows (vault operations, reward claiming, migrations) are executed by an assigned **operator**.

Users must trust that:

- the **operator** will execute operator-only instructions in the user's interest (especially migrations and vault operations) and will not use their privileges to extract value or leave positions in an unsafe/unsupported

state; and

- the operator will not submit malformed account graphs (particularly in `remaining_accounts`-driven CPIs) that desynchronize Hobba state from external provider state.

External protocol dependencies (CPI risk)

Hobba is an orchestration layer and relies on multiple external programs via CPI.

Users must trust that:

- integrated providers (**Kamino KLend**, **JupLend**, and supported vault/farm programs) behave as specified and are not upgraded maliciously;
- provider-side pricing, interest, and accounting are correct; and
- any oracle/TWAP or pricing dependencies used by providers cannot be manipulated to the detriment of Hobba users.

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 Fuzz Testing was utilized to test the correctness and robustness of the system. This approach involves manually guiding the fuzzer by specifying which Solana instruction sequences should be executed and what accounts should be used. The essence of this approach is to ensure that the fuzzer is guided by the intended behavior of the system and not just randomly exploring the codebase. This helps the auditors to focus on the most critical parts of the codebase and not waste time on exploring the codebase randomly.

For this audit revision, pseudo fork testing was utilized to create a testing environment that closely mimics the Solana Mainnet. This technique involves dumping real Solana accounts and programs from the Mainnet (such as on-chain accounts storing asset prices) and loading them into the TridentSVM execution environment. This approach ensures that the fuzz tests interact with realistic account data and program dependencies. The pseudo designation indicates that while we replicate Mainnet state, Trident cannot yet automatically fetch all required accounts from the desired cluster, requiring manual cloning of necessary Solana accounts and programs.

Additionally, property-based fuzz testing was employed to verify the correctness and expected behavior of Solana program instructions. This approach involves writing invariant checks—mathematical properties that must always hold true—which are automatically validated after each instruction executes successfully. These invariants allow us to compare the actual program behavior against expected behavior, catching subtle logic errors and account state inconsistencies. The fuzz testing campaign simulated a comprehensive spectrum of Solana system instructions while enforcing strong assertions about the program's behavior.

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: Migration windup/unwind lack on-chain phase gating, enabling standalone operator-funded migrations and unearned fee extraction

High severity issue

Impact:	High	Likelihood:	Medium
Target:	programs/hobba/src/instructions/operator/migrate/juplend_windup.rs, programs/hobba/src/instructions/operator/migrate/kamino_unwind.rs	Type:	Logic error

Description

The migration flow is designed to be executed as a 4-instruction atomic transaction (flash borrow → unwind → windup → flash repay). However, the migration instructions are exposed as public operator-only entrypoints and do not enforce an on-chain migration phase/state that binds the unwind and windup legs together.

`UserState` contains only the current lending provider selector and the currently-tracked provider accounts (`lending_provider`, `obligation`, `lending_market`), and has no "migration in progress" field/nonce/checkpoint.

As a result, windup instructions can pivot `UserState` to the destination provider using the current `pda_collateral_ata` balance, without proving that the previous provider position was repaid and emptied by the matching unwind step in the same transaction. This can desynchronize Hobba's tracked state from the user's actual provider positions.

The issue is not limited to state desynchronization. The unwind instructions

accept liquidity that is merely present in `operator_usdc_ata` / `pda_usdc_ata`, while the windup instructions compute a fresh borrow amount and then transfer the resulting PDA USDC balance to `operator_usdc_ata` for the expected flash-repay leg. Because Hobba does not verify that a real flash borrow occurred earlier in the transaction nor that a matching flash repay will occur later, an assigned operator can self-fund the unwind with ordinary USDC, call the windup as a standalone instruction, and recover `flash_amount + kamino_flash_fee` back into `operator_usdc_ata`.

For example, `migrate_juplend_windup` bridges the entire PDA USDC balance to `operator_usdc_ata` and pivots `UserState` to the new provider, without proving that the source position was unwound earlier in the same transaction or that a real flash-repay leg exists:

Listing 1. Excerpt from juplend_windup

```
154 // ============ Step 1.5: Bridge all PDA USDC → operator for flash_repay
    ============
155 // The trailing top-level flash_repay takes USDC from operator_usdc_ata.
    Move
156 // everything in pda_usdc_ata back to the operator (PDA-signed transfer).
157 let pda_usdc_balance = read_token_balance(&ctx.accounts.pda_usdc_ata)?;
158 if pda_usdc_balance > 0 {
159     let bridge_ix = build_transfer_ix(
160         ctx.accounts.token_program.key(),
161         ctx.accounts.pda_usdc_ata.key(),
162         ctx.accounts.operator_usdc_ata.key(),
163         ctx.accounts.user_state.key(),
164         pda_usdc_balance,
165     );
166     invoke_signed(
167         &bridge_ix,
168         &[
169             ctx.accounts.pda_usdc_ata.to_account_info(),
170             ctx.accounts.operator_usdc_ata.to_account_info(),
171             ctx.accounts.user_state.to_account_info(),
172             ctx.accounts.token_program.to_account_info(),
173         ],
174         signer_seeds,
175     )?;
176 }
```

```
177
178 // ============ Step 2: Mutate UserState — pivot to JupLend ============
179 let user_state = &mut ctx.accounts.user_state;
180 let from_provider = user_state.lending_provider;
181 user_state.lending_provider = 1;
182 user_state.obligation = ctx.accounts.new_position.key();
183 user_state.lending_market = ctx.accounts.new_vault_config.key();
```

The same pattern applies in the opposite direction (JupLend → Kamino) via `migrate_juplend_unwind` and `migrate_kamino_windup`, which likewise derive borrow sizing from flash parameters and return the resulting PDA USDC to `operator_usdc_ata` before mutating `UserState`.

Exploit scenario

Alice has an active Kamino obligation tracked by Hobba.

1. Bob (the assigned operator, or anyone who compromised the operator key) transfers enough of his own USDC into `operator_usdc_ata` and calls `migrate_kamino_unwind`, repaying Alice's Kamino debt and withdrawing her collateral into `pda_collateral_ata`.
2. Instead of continuing the intended same-transaction flash-loan sequence, Bob later calls `migrate_juplend_windup` as a standalone operator-only instruction.
3. Hobba deposits Alice's collateral into JupLend, borrows `flash_amount + kamino_flash_fee - pda_usdc_balance`, and transfers the resulting PDA USDC balance to `operator_usdc_ata`.
4. Bob recovers his bridge capital and pockets the `kamino_flash_fee` component (or more, if combined with an inflated fee input; see W5), while Alice is left with destination-provider debt and no on-chain proof that a real flash-loan sandwich occurred.

Even without the operator-funded sequence above, standalone windup calls can still pivot `UserState` away from the source provider without proving that

the source position was emptied, leaving the original provider position untracked by Hobba.

Recommendation

Enforce migration atomicity and ordering on-chain.

For example:

- Introduce a migration state machine (fields on `UserState` or a dedicated `MigrationState` PDA) where unwind sets a `migration_phase`/`migration_nonce` and windup requires and clears it before mutating `lending_provider`, `obligation`, and `lending_market`.
- Enforce same-transaction sequencing via `sysvar::instructions` (windup must observe the matching unwind earlier in the same transaction for the same `user_state`, and a matching flash-repay later in the same transaction before any bridge to `operator_usdc_ata` is allowed).
- Derive or validate the flash fee on-chain and require the post-windup transfer target to match the actual flash-repay obligation, rather than blindly returning all PDA USDC to the operator settlement account.

Fix 1.1

The client fixed the issue by removing the affected instructions.

Go back to Findings Summary

M1: claim_vault_rewards does not bind reward destinations nor validate the Associated Token Program, enabling reward redirection and arbitrary CPI invocation

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/hobba/src/instructions/operator/claim_vault_rewards.rs	Type:	Data validation

Description

`claim_vault_rewards` allows an operator to harvest kFarms rewards on behalf of a user. The instruction accepts reward-related accounts via `remaining_accounts` and does not constrain them to canonical Associated Token Accounts (ATAs) for the user.

Additionally, `claim_vault_rewards` takes the `associated_token_program` as an unchecked account and forwards it into an inner CPI that sets the invoked program id to the supplied address.

`claim_vault_rewards` does not constrain `associated_token_program`:

Listing 2. Excerpt from claim_vault_rewards

```
 9 #[derive(Accounts)]
10 pub struct ClaimVaultRewards<'info> {
11     // ============ Common operator check ============
12
13     /// The operator who can manage vault operations
14     #[account(mut)]
15     pub operator: Signer<'info>,
16
17     /// The UserState PDA (acts as signer for farm CPIs)
18     #[account(
```

```
19          seeds = [USER_STATE_SEED, user.key().as_ref(),
    collateral_mint.key().as_ref()],
20          bump = user_state.bump,
21          constraint = user_state.operator == operator.key() @
    HobbaError::UnauthorizedOperator,
22      )]
23      pub user_state: Account<'info, UserState>,
24
25      /// CHECK: The user whose rewards are being claimed
26      pub user: UncheckedAccount<'info>,
27
28      /// CHECK: Collateral mint identifying this position
29      pub collateral_mint: UncheckedAccount<'info>,
30
31      // ============ Farm accounts ============
32
33      /// CHECK: Farm state account — validated by kFarms program
34      #[account(mut)]
35      pub farm_state: UncheckedAccount<'info>,
36
37      /// CHECK: User's farm state account — validated by kFarms program
38      #[account(mut)]
39      pub user_farm_state: UncheckedAccount<'info>,
40
41      /// CHECK: kFarms global config — validated by kFarms program
42      pub global_config: UncheckedAccount<'info>,
43
44      /// CHECK: Farm vaults authority PDA — validated by kFarms program
45      pub farm_vaults_authority: UncheckedAccount<'info>,
46
47      /// CHECK: Scope prices oracle (or farms program ID as placeholder)
48      pub scope_prices: UncheckedAccount<'info>,
49
50      // ============ Programs ============
51
52      /// CHECK: Token program
53      pub token_program: UncheckedAccount<'info>,
54
55      /// CHECK: kFarms program
56      #[account(address = FARMS_PROGRAM_ID)]
57      pub farms_program: UncheckedAccount<'info>,
58
59      /// System program (for ATA creation)
60      pub system_program: Program<'info, System>,
61
62      /// CHECK: Associated Token Program
63      pub associated_token_program: UncheckedAccount<'info>,
```

```
64 }
```

The ATA creation helper uses the caller-supplied `associated_token_program` as the CPI program id, so substituting it can bypass ATA-derivation enforcement.

If the canonical Associated Token Program is provided, supplying a non-canonical `user_reward_ata` causes the ATA creation step to revert, turning the issue into a denial of service.

However, because Hobba does not pin `associated_token_program`, an operator can substitute it with an arbitrary program that returns success without enforcing ATA derivation and thereby bypass the ATA-address check. In that case, the operator-controlled `user_reward_ata` is forwarded into the kFarms harvest CPI as the reward destination.

Exploit scenario

Alice is a user whose vault rewards should be claimed to her reward ATAs. Bob is Alice's assigned operator.

1. Bob constructs a `claim_vault_rewards` transaction and supplies `user_reward_ata` accounts in `remaining_accounts` that are controlled by Bob.
2. Bob supplies a non-ATA program as `associated_token_program` that returns success without enforcing ATA derivation.
3. Bob executes `claim_vault_rewards`.
4. Hobba invokes the supplied `associated_token_program` during the "create ATA" CPI step and then calls the kFarms harvest CPI using the operator-controlled `user_reward_ata` as the reward destination.

Recommendation

Bind reward destinations and programs:

- validate that `associated_token_program` and `token_program` are the expected program ids for Associated Token Account creation and SPL Token operations before performing CPIs; and
- enforce `user_reward_ata == get_associated_token_address(&user.key(), &reward_mint.key())` for each harvested reward.

If non-ATA reward destinations are intentionally supported, update the naming/docs to reflect that these are arbitrary destinations and add explicit mint/authority checks to avoid accidental redirection.

Fix 1.1

The client fixed the issue by (1) constraining `associated_token_program` as `Program<AssociatedToken>` (pinning it to the canonical ATA program id) and (2) enforcing `user_reward_ata == get_associated_token_address(&user.key(), &reward_mint.key())` for each harvested reward.

Go back to Findings Summary

M2: User-facing token accounts are not bound to canonical ATAs, enabling destination substitution in user-signed flows

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	programs/hobba/src/instructions/user/borrow.rs, programs/hobba/src/instructions/user/withdraw.rs	Type:	Data validation

Description

Several user-facing token accounts are declared as `UncheckedAccount` and documented as the user's associated token accounts (ATAs), but Hobba does not locally validate that they are the canonical ATA for `(owner, expected_mint)`.

For example, `borrow` takes a `user_usdc_ata` destination as an `UncheckedAccount` with no ATA constraint:

Listing 3. Excerpt from borrow

```
41 /// CHECK: Validated by token program
42 #[account(mut)]
43 pub pda_usdc_ata: UncheckedAccount<'info>,
44
45 /// The user's USDC ATA (final destination)
46 /// CHECK: Validated by token program
47 #[account(mut)]
48 pub user_usdc_ata: UncheckedAccount<'info>,
49
50 /// Token program
51 /// CHECK: Token program
52 pub token_program: UncheckedAccount<'info>,
```

Similarly, `withdraw` takes a `user_destination_ata` (final destination) as an unchecked account.

This is inconsistent with other flows (e.g., `harvest_vault`) that explicitly bind destinations to canonical ATAs via `get_associated_token_address`.

The missing destination binding is currently limited to user-signed flows. In practice, this means the issue occurs if the user unintentionally signs a transaction that supplies a non-user destination token account (e.g., due to a malicious/compromised frontend or transaction builder), causing the program to transfer assets to an attacker-controlled account.

Exploit scenario

Alice intends to borrow/withdraw to her USDC/collateral ATA.

1. Bob operates (or compromises) a frontend/backend that constructs Alice's transaction.
2. Bob replaces the "user ATA" destination account (e.g., `user_usdc_ata` / `user_destination_ata`) with Bob's token account for the same mint.
3. Alice signs the transaction.
4. Hobba transfers funds to the substituted destination, and Alice receives nothing.

Recommendation

Bind user-facing destination token accounts to canonical ATAs, consistent with `harvest_vault`.

Examples:

- enforce `user_usdc_ata == get_associated_token_address(&owner.key(), &USDC_MINT)` in user borrow/withdraw flows; and

- enforce collateral destinations/sources (where intended) as `get_associated_token_address(&owner.key(), &collateral_mint.key())`.

If non-ATA destinations are intentionally supported, update the account naming/docs to reflect that these accounts are arbitrary user-provided token accounts (not necessarily ATAs) and consider adding explicit mint/owner checks to prevent accidental misbinding.

Fix 1.1

The client fixed the issue by binding user-facing destination token accounts to canonical ATAs (e.g., `user_usdc_ata` in `borrow` and `user_destination_ata` in `withdraw`) via `get_associated_token_address(…)` constraints.

Go back to Findings Summary

M3: JupLend borrow skips lender borrow after partial vault withdrawal

Medium severity issue

Impact:	Low	Likelihood:	High
Target:	programs/hobba/src/instructions/user/borrow.rs	Type:	Logic error

Description

The unified `borrow` instruction first tries to withdraw USDC from the user's vault shares. If the vault does not cover the full requested amount, Hobba should borrow the missing part from the active lending provider.

The Kamino branch borrows the missing amount, but the JupLend branch skips the lender borrow whenever the vault returns any non-zero amount:

Listing 4. Excerpt from borrow

```
200     ctx.accounts.user_state.key(),
201     transfer_amount,
202 );
203
204 invoke_signed(
205     &transfer_ix,
206     &[
207         ctx.accounts.pda_usdc_ata.to_account_info(),
208         ctx.accounts.user_usdc_ata.to_account_info(),
209         ctx.accounts.user_state.to_account_info(),
210         ctx.accounts.token_program.to_account_info(),
211     ],
212     signer_seeds,
213 )?;
214
215 msg!("Transferred {} USDC to user", transfer_amount);
216
217 // ============ Common: Update vault tracking ============
218
219 if vault_usdc_received > 0 {
```

```
220     let user_state = &mut ctx.accounts.user_state;
221     user_state.deposited_to_vault = user_state
222         .deposited_to_vault
223         .saturating_sub(vault_usdc_received);
224 }
```

This means a partial vault withdrawal is treated as if the vault fully covered the borrow. As a result, the instruction can succeed while transferring less USDC than the user requested, breaking expected `borrow(liquidity_amount)` semantics.

Exploit scenario

Alice requests a JupLend-backed borrow of `100` USDC.

1. Her vault withdrawal returns only `20` USDC.
2. The JupLend branch treats any non-zero `vault_usdc_received` as full coverage and skips the lender borrow entirely.
3. The instruction transfers only `20` USDC to Alice even though the borrow request amount was `100`.

Recommendation

For the JupLend branch, compute the missing amount as `liquidity_amount.saturating_sub(vault_usdc_received)` and borrow that remainder from JupLend (if non-zero), mirroring the Kamino behavior.

Acknowledgment 1.1

The issue was acknowledged by the client, with the following comment:

> *Prevented offchain. The operator borrows more than what the UI allows to borrow. I.e. user LTV is always ⇐ [user_borrowed + operator_borrowed] LTV' and because of this invariant borrowing from the UI always has enough liquidity*

> *in the operator borrow part.*
>
> — Hobba team

Go back to Findings Summary

M4: claim_vault_rewards passes UserState as a writable CPI signer while the outer account is read-only

Medium severity issue

Impact:	Low	Likelihood:	High
Target:	programs/hobba/src/instructions/operator/claim_vault_rewards.rs	Type:	Logic error

Description

`ClaimVaultRewards` declares `user_state` without `mut`, so the outer instruction treats the account as read-only:

Listing 5. Excerpt from claim_vault_rewards

```
17 /// The UserState PDA (acts as signer for farm CPIs)
18 #[account(
19     seeds = [USER_STATE_SEED, user.key().as_ref(),
   collateral_mint.key().as_ref()],
20     bump = user_state.bump,
21     constraint = user_state.operator == operator.key() @
   HobbaError::UnauthorizedOperator,
22 )]
23 pub user_state: Account<'info, UserState>,
```

However, the kFarms CPI helper marks the `user_state` account meta as writable signer by using `AccountMeta::new(user_state.key(), true)`:

On Solana, a program cannot escalate an account from read-only in the outer instruction to writable in an inner CPI. As a result, this CPI invocation is expected to fail with a privilege escalation error, preventing `claim_vault_rewards` from successfully harvesting rewards.

Exploit scenario

Bob is the assigned operator for Alice and tries to claim her vault rewards.

1. Bob calls `claim_vault_rewards` with otherwise-correct farm accounts.
2. Hobba enters the kFarms CPI with `user_state` passed read-only in the outer instruction but writable in the inner account metas.
3. The inner CPI fails with a writable privilege escalation error, so reward claiming is blocked.

Recommendation

Make `user_state` mutable in `ClaimVaultRewards` if the Farms CPI requires it to be writable:

```
#[account(mut, …)] pub user_state: Account<'info, UserState>
```

Alternatively, if Farms does not require writable access to `user_state`, update the CPI helper to use `AccountMeta::new_readonly(user_state.key(), true)` so the inner CPI does not request write privilege.

Fix 1.1

The client fixed the issue by marking `user_state` as mutable in `ClaimVaultRewards`, matching the kFarms CPI requirement for a writable signer and preventing the writable privilege escalation failure.

Go back to Findings Summary

L1: Kamino and JupLend position accounting is read without required refresh, causing stale repay sizing and incorrect health assumptions

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	programs/hobba/src/instructions/user/kamino/prepare_withdraw.rs, programs/hobba/src/instructions/user/juplend/prepare_withdraw.rs, programs/hobba/src/instructions/operator/juplend/harvest_vault.rs	Type:	Data validation

Description

Several Hobba instructions read external lending provider state to size repayments and sanity-check invariants, but do so without ensuring the provider state is refreshed to reflect interest accrual.

Kamino

Kamino interest accrues through reserve state, and the obligation's per-borrow `borrowed_amount_sf` and per-deposit exchange-rate derived values are brought up to date via `refresh_reserve` and `refresh_obligation`. Hobba's readers are pure byte-parsers:

- `read_kamino_obligation_debt` sums `borrowed_amount_sf / 2^60` from obligation borrow entries; and

- `read_kamino_collateral_native` converts obligation cToken amounts to underlying using reserve exchange rates.

As a result, reading these values before refresh can under-report debt and/or mis-estimate collateral.

This pattern appears in:

- `prepare_withdraw` repay sizing: `compute_repay_amount` reads obligation debt and collateral before any refresh CPI and uses them to compute proportional repay.
- `harvest_vault` repay sizing: the Kamino harvest path reads `total_debt` without refreshing the obligation first, then uses it to compute `user_debt` and cap `repay_amount` (it refreshes only the USDC reserve immediately before repay CPI).
- `migrate_kamino_unwind` pre-refresh sanity: `debt_before` is read before refresh and used to gate `FlashLoanShortfall` checks.

JupLend

JupLend debt/collateral readers depend on `vault_state` exchange prices. Hobba reads these exchange prices directly from the `vault_state` account and uses them to compute debt/collateral.

However, the JupLend harvest path explicitly forces an on-chain exchange price update via `invoke_update_exchange_prices` before reading debt, noting that direct readers otherwise see stale values. Other flows still read position/vault_state without performing this update.

This pattern appears in:

- `prepare_withdraw` (JupLend) repay sizing: reads debt/collateral from `position + vault_state` without updating exchange prices first.

- `migrate_juplend_unwind` (operator migration) debt/collateral reads: reads debt/collateral from `position + vault_state` without updating exchange prices first.

While the eventual Kamino/JupLend CPIs generally enforce health and correctness for the actual repay/withdraw, these stale reads can cause Hobba to mis-size repayments, mis-predict post-operation health, and fail transactions unexpectedly.

Exploit scenario

Alice has a position whose debt has accrued interest but the provider state used by Hobba's reader is stale (Kamino obligation not refreshed; or JupLend exchange prices in `vault_state` not updated).

1. Alice calls `prepare_withdraw` with a large `collateral_to_withdraw`.
2. Hobba under-estimates the proportional repay amount because it reads stale debt/collateral data.
3. Hobba repays only the under-sized amount.
4. Alice calls `withdraw` expecting the target LTV to be maintained, but the lender-side withdraw (Kamino withdraw CPI, or JupLend operate-based withdraw) rejects the collateral withdrawal due to insufficient repayment / health constraints, reverting the transaction.

In operator flows, stale debt reads can also mis-size repay amounts (e.g., in `harvest_vault` debt splitting logic), causing less debt repayment than expected and leaving the position at a higher LTV than intended after the instruction.

Recommendation

Ensure provider state is refreshed before any local reads that influence repay sizing, sanity checks, or debt splitting.

Examples:

- For Kamino, refresh the relevant reserves and obligation before reading debt/collateral for computations (or move the computation to after refresh and use refreshed accounts).
- For JupLend, update exchange prices (or otherwise ensure `vault_state` is current) before using `vault_state` values in `read_juplend_debt_usdc` / `read_juplend_collateral_native`. Align all reads with the pattern already used in the JupLend harvest path.

If refreshing is too expensive for some entrypoints, avoid making safety-critical decisions based on direct reads of potentially stale provider accounts.

Acknowledgment 1.1

The issue was acknowledged by the client.

Go back to Findings Summary

L2: claim_vault_rewards does not validate the Associated Token Program before invoking ATA creation CPI

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	programs/hobba/src/instructions/operator/claim_vault_rewards.rs	Type:	Data validation

Description

Hobba is inconsistent about validating the Associated Token Program account before performing ATA-creation CPIs.

Most ATA creation paths constrain `associated_token_program` to the canonical Associated Token Program ID via:

```
#[account(address = ATA_PROGRAM_ID)]
```

However, `claim_vault_rewards` declares `associated_token_program` as an `UncheckedAccount` without an address constraint:

Listing 6. Excerpt from claim_vault_rewards

```
59     /// System program (for ATA creation)
60     pub system_program: Program<'info, System>,
61
62     /// CHECK: Associated Token Program
63     pub associated_token_program: UncheckedAccount<'info>,
64 }
```

The helper `invoke_create_ata_idempotent` then uses the supplied `associated_token_program` as the CPI target program id.

As a result, `claim_vault_rewards` can be made to invoke an arbitrary program in place of the Associated Token Program. This is inconsistent with other direct ATA-creation paths in Hobba, which validate the Associated Token Program id.

Exploit scenario

Alice is the protocol operator calling `claim_vault_rewards` for a user.

1. Bob provides (or causes Alice's transaction builder to include) a malicious program account in place of the canonical Associated Token Program for `associated_token_program`.
2. Alice signs and submits the transaction.
3. Hobba invokes the malicious program as part of `invoke_create_ata_idempotent`, with the operator marked as a signer and payer in the CPI account list.
4. The malicious program can execute unintended logic (e.g., attempt to misuse the signer/payer context), or at minimum cause unexpected behavior/DoS of reward claiming.

Recommendation

In `ClaimVaultRewards`, constrain `associated_token_program` to the canonical ATA program id:

```
#[account(address = <ATA_PROGRAM_ID>)]
```

Optionally, also constrain `token_program` to the expected SPL Token program id for defense-in-depth, since it is also forwarded into the ATA creation CPI.

Fix 1.1

The client fixed the issue by changing `associated_token_program` to `Program<AssociatedToken>` in `ClaimVaultRewards`, which pins it to the canonical Associated Token Program id before performing ATA-creation CPIs.

Go back to Findings Summary

L3: set_treasury allows misconfiguration that can block harvest_vault fee transfers

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	programs/hobba/src/instructions/admin/set_treasury.rs	Type:	Data validation

Description

`set_treasury` accepts any `Pubkey` as the new treasury and does not validate that the address is non-default or otherwise "usable" for receiving USDC fees.

Listing 7. Excerpt from set_treasury

```
#[derive(Accounts)]
pub struct SetTreasury<'info> {
    pub deployer: Signer<'info>,

    #[account(
        mut,
        seeds = [PROGRAM_CONFIG_SEED],
        bump = program_config.bump,
    )]
    pub program_config: Account<'info, ProgramConfig>,

    /// The program's data account — contains the upgrade authority
    #[account(
        seeds = [crate::ID.as_ref()],
        bump,
        seeds::program = anchor_lang::solana_program::bpf_loader_upgradeable::id(),
        constraint = program_data.upgrade_authority_address == Some(deployer.key()) @ HobbaError::UnauthorizedDeployer,
    )]
    pub program_data: Account<'info, ProgramData>,
}

pub fn handler(ctx: Context<SetTreasury>, new_treasury: Pubkey) -> Result<()> {
```

```
29      ctx.accounts.program_config.treasury = new_treasury;
30      msg!("Treasury changed to: {}", new_treasury);
31      Ok(())
```

Downstream, `harvest_vault` requires `treasury_usdc_ata` to be the canonical USDC ATA for `program_config.treasury`. If the configured treasury is unusable or the canonical USDC ATA does not exist, then any harvest that computes a non-zero `fee_amount` will fail when attempting the fee transfer, blocking the harvest transaction.

This issue requires privileged misconfiguration: `set_treasury` is gated by the program upgrade authority.

Exploit scenario

Alice is the upgrade authority (or an admin acting with upgrade authority).

1. Alice calls `set_treasury` with an incorrect treasury (e.g., `Pubkey::default()` or a key whose canonical USDC ATA is not created).
2. Later, Bob (the operator) calls `harvest_vault` for a user and the harvest computes a non-zero `fee_amount`.
3. The transaction fails when attempting to transfer the fee to the required canonical `treasury_usdc_ata`, preventing harvest execution until the treasury is fixed.

Recommendation

Add validation in `set_treasury`, e.g.:

- reject `Pubkey::default()`; and
- optionally require the canonical USDC ATA for the new treasury to exist (or provide an admin/operator flow to create it idempotently).

Alternatively, make `harvest_vault` create the treasury ATA idempotently (with

a validated Associated Token Program) before transferring fees.

Fix 1.1

The client fixed the issue by rejecting `Pubkey::default()` as a treasury, validating that `treasury_usdc_ata` is the canonical USDC ATA for the new treasury, and creating that ATA idempotently during `set_treasury`.

Go back to Findings Summary

L4: migrate_juplend_unwind does not verify collateral is fully withdrawn

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	programs/hobba/src/instructions/operator/migrate/juplend_unwind.rs	Type:	Logic error

Description

`migrate_juplend_unwind` intends to fully repay JupLend debt and withdraw all JupLend collateral into `pda_collateral_ata` before switching the user to Kamino.

The instruction computes `collateral_native` from the JupLend `position` and `vault_state` (using a caller-supplied `collateral_decimals`) and passes `-collateral_native` into the JupLend `operate` CPI to withdraw collateral. After the CPI, it verifies only that the debt is zero:

Listing 8. Excerpt from juplend_unwind

```
107 // Read current position size to compute the operate deltas.
108 let collateral_native =
109     read_juplend_collateral_native(&ctx.accounts.position,
    &ctx.accounts.vault_state, collateral_decimals)?;
110 let debt_usdc = read_juplend_debt_usdc(&ctx.accounts.position,
    &ctx.accounts.vault_state)?;
111 require!(collateral_native > 0, HobbaError::ZeroAmount);
112
113 // Sanity: pda_usdc_ata must hold ≥ debt + 1 (JupLend rounding) after the
    bridge transfer.
114 let pda_usdc_balance = read_token_balance(&ctx.accounts.pda_usdc_ata)?;
115 require!(pda_usdc_balance >= debt_usdc.saturating_add(1),
    HobbaError::FlashLoanShortfall);
116
117 let user_key = ctx.accounts.user.key();
```

```
let collateral_mint_key = user_state.collateral_mint;
let bump = user_state.bump;
let seeds = &[
    USER_STATE_SEED,
    user_key.as_ref(),
    collateral_mint_key.as_ref(),
    &[bump],
];
let signer_seeds = &[&seeds[..]];

let collateral_before =
read_token_balance(&ctx.accounts.pda_collateral_ata)?;

// operate(-col, -debt): repay full debt and withdraw all collateral
atomically.
// pda_collateral_ata is the supply-token (collateral) account, pda_usdc_ata
is the borrow-token (USDC) account.
invoke_operate(
    user_state.to_account_info(),
    ctx.accounts.pda_collateral_ata.to_account_info(),
    ctx.accounts.pda_usdc_ata.to_account_info(),
    juplend_program.clone(),
    operate_accounts,
    -(collateral_native as i128),
    -(debt_usdc as i128),
    None,
    &[1, 1, 2],
    signer_seeds,
)?;

let collateral_after =
read_token_balance(&ctx.accounts.pda_collateral_ata)?;
let collateral_received = measure_balance_delta(collateral_before,
collateral_after)?;

// Sanity: JupLend position should be empty after a full unwind.
let debt_remaining = read_juplend_debt_usdc(&ctx.accounts.position,
&ctx.accounts.vault_state)?;
require!(debt_remaining == 0, HobbaError::MigrationDebtRemaining);
```

There is no equivalent post-CPI check that the JupLend position collateral is fully withdrawn (i.e., that the position is empty on the collateral side).

If `collateral_native` is computed incorrectly due to rounding, stale `vault_state`, or an incorrect `collateral_decimals` input, the unwind may leave

residual collateral in the source JupLend position while Hobba proceeds to the next migration steps.

Exploit scenario

Alice is being migrated from JupLend to Kamino.

1. Bob supplies stale sizing inputs or an incorrect `collateral_decimals` value for `migrate_juplend_unwind`.
2. Hobba repays the debt successfully, so the existing post-CPI debt check passes.
3. Because no collateral-empty check exists, some collateral can remain stranded in the old JupLend position while the migration continues toward Kamino.

Recommendation

After the `operate(-col, -debt)` CPI, explicitly verify that remaining collateral is zero (or below a dust threshold) by re-reading the position's collateral amount from `position` + `vault_state`, similar to the existing debt remaining check.

Fix 1.1

The issue was fixed by removing the affected instruction.

Go back to Findings Summary

L5: deposit does not bind supply_token_mint to user_state collateral mint

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	programs/hobba/src/instructions/user/deposit.rs	Type:	Data validation

Description

The `Deposit` accounts struct receives both a `collateral_mint` (used to derive the `UserState` PDA) and a separate `supply_token_mint` (used for token account creation and token movement), but it does not validate that `supply_token_mint` matches the collateral mint recorded in `UserState`.

Listing 9. Excerpt from deposit

```
/// The UserState PDA
#[account(
    mut,
    seeds = [USER_STATE_SEED, owner.key().as_ref(),
collateral_mint.key().as_ref()],
    bump = user_state.bump,
    has_one = owner @ HobbaError::InvalidOwner,
)]
pub user_state: Account<'info, UserState>,

/// CHECK: Collateral mint identifying this position
pub collateral_mint: UncheckedAccount<'info>,

/// The user's cbBTC token account (source)
/// CHECK: Validated by token program
#[account(mut)]
pub user_source_liquidity: UncheckedAccount<'info>,

/// The PDA's cbBTC token account (intermediate for CPI)
/// CHECK: Created if needed, validated by token program
#[account(mut)]
pub pda_source_liquidity: UncheckedAccount<'info>,
```

```
50
51 /// cbBTC mint
52 /// CHECK: Validated by token program
53 pub supply_token_mint: UncheckedAccount<'info>,
```

`supply_token_mint` is then used when creating the PDA's token account and when moving tokens, without a local check that it matches `user_state.collateral_mint`.

There is no local check such as:

`supply_token_mint.key() == user_state.collateral_mint`

or binding `pda_source_liquidity` to the canonical ATA for `(user_state, user_state.collateral_mint)`.

This can lead to confusing or inconsistent behavior if a user (or a malicious transaction builder) supplies mismatched mints/accounts.

Exploit scenario

Alice deposits through a malicious or buggy transaction builder.

1. The builder derives the `UserState` PDA using the correct `collateral_mint`, but supplies a different `supply_token_mint` and matching token accounts in the instruction.
2. Hobba accepts the account set because it never checks that `supply_token_mint == user_state.collateral_mint`.
3. The transaction can proceed with confusing mismatched mint assumptions or fail only later in downstream logic, making the deposit path brittle and easier to misuse.

Recommendation

Bind the supply mint to the position's collateral mint. For example:

- require `supply_token_mint.key() == user_state.collateral_mint`; and
- require `pda_source_liquidity` to equal the canonical ATA for `(user_state, user_state.collateral_mint)`.

Fix 1.1

The client fixed the issue by binding `supply_token_mint` to `user_state.collateral_mint` and constraining `pda_source_liquidity` to the canonical ATA for `(user_state, user_state.collateral_mint)`.

Go back to Findings Summary

L6: Vault flows have inconsistent min-out / slippage enforcement (Perena)

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	programs/hobba/src/instructions/operator/juplend/operator_borrow.rs	Type:	Logic error

Description

Several vault flows pass `0` as the minimum acceptable output when interacting with vault providers. For Perena, these `min_*` parameters are meaningful slippage bounds (they are forwarded into the Perena CPI), so passing `0` disables on-chain min-out protection and shifts safety to off-chain builders and operational assumptions.

For example, `operator_borrow` deposits into the vault with `min_shares_out = 0`:

Listing 10. Excerpt from operator_borrow

```
// ============ Step 2: Deposit USDC into Vault ============

let shares_before = read_token_balance(&accounts.pda_shares_ata)?;

dispatch_vault_deposit(
    vault_provider,
    user_state.to_account_info(),
    accounts.vault.to_account_info(),
    accounts.shares_mint.to_account_info(),
    accounts.pda_usdc_ata.to_account_info(),
    accounts.pda_shares_ata.to_account_info(),
    accounts.usdc_mint.to_account_info(),
    accounts.token_program.to_account_info(),
    accounts.shares_token_program.to_account_info(),
    vault_remaining,
    liquidity_amount,
    0, // min_shares_out: for Perena slippage, computed off-chain
```

```
73     signer_seeds,
74 )?;
```

Similarly, `operator_repay`, `harvest_vault`, and user flows route withdrawals with `min_yielding_out = 0`.

By contrast, `switch_vault` computes and forwards a non-zero floor, providing on-chain slippage protection.

This inconsistency makes it harder to reason about safety guarantees and can expose users/operators to unfavorable vault execution outcomes.

Exploit scenario

Alice has funds managed in a Perena-backed vault path.

1. Bob submits an operator or user flow that routes through Perena while the effective vault exchange rate is unfavorable.
2. Because Hobba passes `0` as the minimum acceptable output, the Perena CPI is allowed to settle at any non-negative amount.
3. Alice receives materially fewer shares or less USDC than expected, with no on-chain minimum-out guard to stop the transaction.

Recommendation

Adopt a consistent slippage/min-out strategy across vault flows:

- compute and enforce `min_shares_out` / `min_yielding_out` on-chain when possible (as in `switch_vault`), or
- require the caller to supply explicit min-out parameters and validate them against a reasonable bound, rather than hardcoding `0`.

Fix 1.1

The client fixed the issue by computing and forwarding non-zero Perena

slippage floors on-chain (via `compute_min_shares_out` / `compute_min_yielding_out`) across vault deposit/withdraw flows instead of hardcoding `0`.

Go back to Findings Summary

L7: JupLend flows do not bind operate account list to UserState fields, allowing state/CPI desynchronization

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/hobba/src/juplend_cpi.rs, programs/hobba/src/instructions/operator/migrate/juplend_windup.rs	Type:	Data validation

Description

For JupLend users, Hobba stores the expected JupLend position and vault config in `UserState`:

- `user_state.obligation` (JupLend position)
- `user_state.lending_market` (JupLend vault config)

During `initialize_user` (JupLend path), these values are taken directly from caller-supplied `remaining_accounts`:

Listing 11. Excerpt from initialize_user

```
114         // Kamino needs 2 CPIs (init_user_metadata + init_obligation)
115         initialize_user_kamino::handler(&ctx.accounts, lender_remaining,
    bump)?;
116     }
117     1 => {
118         // JupLend: remaining[0]=position, remaining[1]=vault_config
119         user_state.obligation = lender_remaining[0].key();
120         user_state.lending_market = lender_remaining[1].key();
121
122         msg!(
123             "JupLend user initialized with position: {}",
```

```
            user_state.obligation
        );
    }
    _ => return Err(HobbaError::InvalidLendingProvider.into()),
};

emit!(UserInitialized {
    user: owner_key,
    collateral_mint: collateral_mint_key,
```

However, subsequent JupLend `operate` CPIs do not consult `user_state.obligation` / `user_state.lending_market` and do not validate that the supplied operate account list matches them. Hobba forwards the caller-provided `remaining_accounts` into the CPI wrapper as-is.

This is particularly relevant in migration, where `migrate_juplend_windup` stores the named destination position/config only after the CPI succeeds, but the CPI itself is driven by `remaining_accounts`.

But the actual JupLend vault config and position used by the `operate` CPI are provided through `remaining_accounts`, not derived from `new_position` /`new_vault_config` nor checked against them. This can result in Hobba recording one position/config in `UserState` while the CPI operated on a different one, breaking invariants and potentially leaving debt/collateral untracked by Hobba.

Exploit scenario

Alice is migrating to JupLend.

1. Bob (the operator / transaction builder) supplies a valid `new_position` and `new_vault_config` as named accounts (so Hobba will store them).
2. In `remaining_accounts`, Bob supplies a different JupLend `operate` account graph that points to a different position/config.
3. The JupLend CPI operates on the accounts in `remaining_accounts`, but

Hobba stores the named `new_position`/`new_vault_config`.

4. Alice's resulting debt/collateral can be created in a position that Hobba does not track in `UserState`.

Recommendation

Bind JupLend operate account lists to `UserState` before CPI:

- Require the operate account list's vault config and position to equal `user_state.lending_market` and `user_state.obligation` (or the named destination accounts in migration).
- Where feasible, avoid caller-supplied "position/config" duplication by deriving the critical accounts from `UserState` and using them directly.

Fix 1.1

The client fixed the issue by binding JupLend `operate` account lists to `UserState` (e.g., validating that the operate list's vault config and position match `user_state.lending_market` / `user_state.obligation`, and using helpers that splice named `vault_config`/`position` into the operate list). The migration windup instruction referenced in this finding was also removed.

Go back to Findings Summary

L8: switch_vault accepts a non-canonical source share account and can strand old vault shares

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/hobba/src/instructions/operator/switch_vault.rs	Type:	Logic error

Description

`switch_vault` constrains the destination share account `new_pda_shares_ata` to the canonical ATA for `(user_state, new_shares_mint)`, but it does not apply the same constraint to the old/source share account `old_pda_shares_ata`.

Listing 12. Excerpt from switch_vault

```
40 // --- Old Vault (for withdrawal) ---
41
42 /// The current/old vault account
43 /// CHECK: Must match user_state.current_vault
44 #[account(
45     mut,
46     constraint = old_vault.key() == user_state.current_vault @
   HobbaError::VaultMismatch
47 )]
48 pub old_vault: UncheckedAccount<'info>,
49
50 /// The old vault shares mint
51 /// CHECK: Validated by vault program
52 #[account(mut)]
53 pub old_shares_mint: UncheckedAccount<'info>,
54
55 /// The user's PDA shares ATA for old vault
56 /// CHECK: Validated by token program
57 #[account(mut)]
58 pub old_pda_shares_ata: UncheckedAccount<'info>,
59
60 // --- New Vault (for deposit) ---
61
62 /// The new vault account
```

```
63 /// CHECK: Will be set as new vault
64 #[account(mut)]
65 pub new_vault: UncheckedAccount<'info>,
66
67 /// The new vault shares mint
68 /// CHECK: Validated by vault program
69 #[account(mut)]
70 pub new_shares_mint: UncheckedAccount<'info>,
71
72 /// The user's PDA shares ATA for new vault
73 /// CHECK: Must be the canonical ATA of user_state for new_shares_mint, so
   newly
74 /// minted vault shares cannot be redirected to an operator-controlled
   account.
75 #[account(
76     mut,
77     constraint = new_pda_shares_ata.key() ==
   get_associated_token_address(&user_state.key(), &new_shares_mint.key()) @
   HobbaError::InvalidOwner
78 )]
79 pub new_pda_shares_ata: UncheckedAccount<'info>,
```

The instruction then reads `shares_to_withdraw` directly from the supplied `old_pda_shares_ata` and withdraws that amount from the old vault.

After depositing into the new vault, `switch_vault` unconditionally updates `user_state.current_vault` and `user_state.vault_provider` to the new vault, without verifying that the canonical old-vault share account was used or that all old shares were withdrawn.

As a result, an operator can switch state using a small auxiliary share account (controlled by the `UserState` PDA) while leaving the user's real old-vault shares in the canonical ATA. Once `current_vault` points to the new vault, normal Hobba vault flows operate on the new vault and cannot reach the stranded old-vault shares through the standard path.

Exploit scenario

Alice has vault shares in the canonical ATA for the current vault.

1. Bob (the operator) creates or finds another token account for the same old shares mint whose authority is the `UserState` PDA.
2. Bob transfers a small amount of old vault shares into that auxiliary account.
3. Bob calls `switch_vault`, passing the auxiliary account as `old_pda_shares_ata`.
4. Hobba withdraws only the auxiliary account's shares, deposits the resulting USDC into the new vault, and updates `user_state.current_vault` to the new vault.
5. Alice's real old vault shares remain in the canonical old-vault ATA, but Hobba now points to the new vault, making the old shares effectively stranded from the normal flow.

Recommendation

Bind `old_pda_shares_ata` to the canonical ATA for `(user_state, old_shares_mint)` using the same model used for `new_pda_shares_ata`.

Optionally, after the old-vault withdrawal, assert that the canonical old-vault shares ATA balance is zero (or below a dust threshold) before updating `user_state.current_vault`.

Fix 1.1

The client fixed the issue by constraining `old_pda_shares_ata` to the canonical ATA for `(user_state, old_shares_mint)` (matching the destination ATA constraints), preventing auxiliary share-account substitution that could strand old vault shares.

Go back to Findings Summary

L9: Perena vault CPIs do not bind vault_remaining to the named vault and mints

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	programs/hobba/src/vault_dispatch.rs	Type:	Data validation

Description

Hobba's vault dispatcher for Perena builds the CPI instruction entirely from the raw `vault_remaining` account list. It only binds two operator-controlled slots (the PDA USDC account and the PDA shares account) to the named accounts, but it does not validate that the Perena `vault_state` (slot 1), USDC mint (slot 4), and share mint (slot 5) match the named `vault`, `usdc_mint`, and `shares_mint` accounts used elsewhere for state transitions and accounting.

Perena deposit dispatch only checks slots 6 and 7:

Listing 13. Excerpt from vault_dispatch

```
93 VAULT_PROVIDER_PERENA => {
94     // Bind operator-controlled slots in vault_remaining to the named PDA
   accounts.
95     // slot 6 = user USDC (source), slot 7 = user shares (destination).
96     // Without these an operator can pass accounts they own and redirect
   either the
97     // newly minted shares or the USDC source.
98     require!(vault_remaining.len() >= 16, HobbaError::PerenaCpiFailed);
99     require_keys_eq!(vault_remaining[6].key(), pda_usdc_ata.key(),
   HobbaError::InvalidOwner);
100     require_keys_eq!(vault_remaining[7].key(), pda_shares_ata.key(),
   HobbaError::InvalidOwner);
101
102     perena_cpi::invoke_perena_deposit(
103         vault_remaining,
104         amount,
105         min_shares_out,
```

```
106         signer_seeds,
107     )
```

Perena withdraw dispatch applies the same slot 6–7-only binding.

As a result, an operator can supply `vault_remaining` corresponding to a different Perena vault than the named `vault` account. This breaks the invariant behind `user_state.current_vault`: Hobba can read and store vault A, while the Perena CPI deposits into or withdraws from vault B.

This mismatch is particularly problematic because Hobba uses the named `vault` to:

- set and enforce `user_state.current_vault` for later flows; and
- parse Perena share pricing inputs (share supply + price) from the named `vault` for repay sizing, profit/fee logic, and expected-output checks.

Exploit scenario

Alice is a user with `user_state.current_vault` set to vault A.

1. Bob (the operator) calls an operator vault flow (e.g., `operator_borrow`, `operator_repay`, or `harvest_vault`) and supplies the named `vault` as vault A.
2. In `vault_remaining`, Bob supplies Perena CPI accounts for a different vault B (including a different Perena vault_state at slot 1 and different mints at slots 4–5), while keeping slots 6–7 bound to Hobba's PDA token accounts.
3. Hobba's accounting uses the named vault A state, but the Perena CPI operates on vault B, causing `user_state.current_vault` and the PDA share balance to become inconsistent.

Recommendation

For Perena vault dispatch, additionally bind the Perena CPI accounts to the named accounts, at minimum:

- `vault_remaining[0]` (Perena depositor/signer PDA) must equal `user_state`;
- `vault_remaining[1]` (vault_state) must equal the named `vault` account;
- `vault_remaining[4]` (USDC mint) must equal the named `usdc_mint`; and
- `vault_remaining[5]` (shares mint) must equal the named `shares_mint`.

Additionally, validate that Perena CPI program and SPL program accounts in `vault_remaining` are the expected program ids, and validate that the named `vault` has the expected owner/account type/layout before parsing it for share price and supply.

Fix 1.1

The client fixed the issue by binding Perena `vault_remaining` to the named Accounts-struct values via `perena_cpi::validate_perena_accounts(…)` in both deposit and withdraw dispatch paths (including vault_state, mints, PDA token accounts, and program ids).

Go back to Findings Summary

W1: Hardcoded third-party account parsing lacks account-type validation, making integrations fragile to wrong accounts and upstream layout changes

Impact:	Warning	Likelihood:	N/A
Target:	programs/hobba/src/token_utils.rs	Type:	Data validation

Description

Hobba parses multiple third-party accounts directly via hardcoded byte offsets and/or implicit ordering assumptions (e.g., Kamino `Obligation`/`Reserve`, KVault `VaultState`, JupLend `position`/`vault_state`, Perena state accounts). These parsed values are then used for control flow and accounting (repay sizing, profit estimation, fee math, and sanity checks).

Most of these reads do not validate that the supplied account data is actually the expected account type for the deployed version of the external protocol. In particular, the program generally does not:

- validate Anchor discriminators for external Anchor accounts;
- validate that the account owner program matches the expected external program; or
- validate a stable version/magic field and minimum data size before applying offsets for non-Anchor accounts.

This means that if the account set is mis-specified (especially in `remaining_accounts` driven flows) or if an upstream protocol changes an account layout, Hobba may silently interpret unrelated bytes as debt, collateral, share supply, or price.

The issue is currently mitigated in some paths by later CPI checks that enforce protocol invariants, but the offset reads still influence Hobba's internal decisions and can lead to incorrect behavior or unexpected transaction failures.

Recommendation

Validate account types before parsing.

Examples:

- For external Anchor accounts, require the correct program owner and verify the first 8 bytes match the expected discriminator from the upstream IDL.
- For non-Anchor accounts, require the correct program owner and validate a stable version/magic field and data length before parsing.
- Prefer upstream interface crates or generated layouts over hardcoded offsets where feasible.

Apply similar validation for SPL Token mints and token accounts used as critical inputs: bind to canonical ATAs or explicitly validate mint/owner/authority fields instead of treating them as generic unchecked accounts.

Partial solution 1.1

The client partially fixed the issue by adding owner/discriminator/min-length validation for externally parsed Anchor accounts (Kamino obligations/reserves, KVault state, JupLend position/vault_state) before reading hardcoded offsets. Perena state parsing remains validated only by owner and minimum length, without strict length/discriminator check.

Go back to Findings Summary

W2: External account readers truncate u128 values to u64 via `as u64`, which can silently wrap at 2^64

Impact:	Warning	Likelihood:	N/A
Target:	programs/hobba/src/token_utils.rs	Type:	Logic error

Description

Several helper readers compute intermediate values in `u128` (e.g., summing scaled-fraction debt or converting cTokens → underlying) and then return them as `u64` using `as u64`.

In Rust, `x as u64` truncates (wraps modulo 2^64) and silently discards the most-significant bits if `x >= 2^64`. This can turn an extremely large computed debt/collateral/share value into an unrelated small `u64`, causing Hobba to mis-size repayments/withdrawals, miscompute health assumptions, and/or take the wrong control-flow branch based on the parsed value.

For example, `read_kamino_obligation_debt` sums debt in a `u128` and then returns `Ok(total as u64)`:

Listing 14. Excerpt from token_utils

```
107 /// Read total borrow debt from a Kamino obligation account.
108 /// Sums all non-empty borrow entries (borrowed_amount_sf / 2^60).
109 pub fn read_kamino_obligation_debt(account: &AccountInfo) -> Result<u64> {
110     let data = account.try_borrow_data()?;
111     let mut total: u128 = 0;
112 
113     for i in 0..KAMINO_MAX_BORROW_ENTRIES {
114         let entry_offset = KAMINO_BORROWS_OFFSET + i *
    KAMINO_BORROW_ENTRY_SIZE;
115         // First 32 bytes of entry is the reserve pubkey; all-zero means
    empty slot
116         if entry_offset + 32 > data.len() {
```

```
            break;
        }
        if data[entry_offset..entry_offset + 32].iter().all(|&b| b == 0) {
            continue;
        }
        let sf_offset = entry_offset + KAMINO_BORROWED_AMOUNT_SF_OFFSET;
        if sf_offset + 16 > data.len() {
            continue;
        }
        let borrowed_sf = u128::from_le_bytes(data[sf_offset..sf_offset +
16].try_into().unwrap());
        total += borrowed_sf / SCALE_FACTOR;
    }

    Ok(total as u64)
}
```

Similar `as u64` truncations exist in other readers (e.g., Kamino collateral conversion, JupLend debt/collateral readers, and some vault AUM/value computations).

While reaching 2^64 in these units is unlikely in typical deployments, this is a brittle parsing pattern: if a protocol upgrade, wrong account, or extreme position size ever causes a parsed intermediate to exceed `u64::MAX`, Hobba will not fail fast and will instead proceed with a wrapped value.

Recommendation

Avoid silent truncation when converting parsed/computed `u128` values to `u64`.

Fix 1.1

The client fixed the issue by introducing `u128_to_u64` and using checked conversions for all u128 → u64 narrowing in external account readers, failing fast on overflow instead of silently truncating via `as u64`.

Go back to Findings Summary

W3: Kamino collateral reader sums all deposit entries but converts using a single reserve exchange rate

Impact:	Warning	Likelihood:	N/A
Target:	programs/hobba/src/token_utils.rs	Type:	Logic error

Description

`read_kamino_collateral_native` parses a Kamino obligation by iterating over all deposit entries and summing the deposited cToken amounts for every non-empty slot. It does not filter deposit entries by their reserve pubkey. The function then converts the total cToken sum to underlying units using the exchange rate derived from a single `reserve` account passed into the function.

Listing 15. Excerpt from token_utils

```
162 /// Read total deposited collateral in native units from a Kamino obligation
    + reserve.
163 /// Reads cToken amount from obligation, converts to underlying via reserve
    exchange rate.
164 pub fn read_kamino_collateral_native(obligation: &AccountInfo, reserve:
    &AccountInfo) -> Result<u64> {
165     let ob_data = obligation.try_borrow_data()?;
166     let mut total_ctokens: u128 = 0;
167 
168     for i in 0..KAMINO_MAX_DEPOSIT_ENTRIES {
169         let entry_offset = KAMINO_DEPOSITS_OFFSET + i *
    KAMINO_DEPOSIT_ENTRY_SIZE;
170         if entry_offset + 32 > ob_data.len() {
171             break;
172         }
173         if ob_data[entry_offset..entry_offset + 32].iter().all(|&b| b == 0)
    {
174             continue;
175         }
176         let amt_offset = entry_offset + KAMINO_DEPOSITED_AMOUNT_OFFSET;
```

```
        if amt_offset + 8 > ob_data.len() {
            continue;
        }
        total_ctokens += u64::from_le_bytes(ob_data[amt_offset..amt_offset + 8].try_into().unwrap()) as u128;
    }

    if total_ctokens == 0 {
        return Ok(0);
    }

    // Convert cTokens to underlying: underlying = cTokens * totalLiquidity / cTokenSupply
    let res_data = reserve.try_borrow_data()?;
    let available = u64::from_le_bytes(
        res_data[KAMINO_RESERVE_AVAILABLE_OFFSET..KAMINO_RESERVE_AVAILABLE_OFFSET + 8]
            .try_into().unwrap(),
    ) as u128;
    let borrowed_sf = u128::from_le_bytes(
        res_data[KAMINO_RESERVE_BORROWED_SF_OFFSET..KAMINO_RESERVE_BORROWED_SF_OFFSET + 16]
            .try_into().unwrap(),
    );
    let ctoken_supply = u64::from_le_bytes(
        res_data[KAMINO_RESERVE_CTOKEN_SUPPLY_OFFSET..KAMINO_RESERVE_CTOKEN_SUPPLY_OFFSET + 8]
            .try_into().unwrap(),
    ) as u128;

    if ctoken_supply == 0 {
        return Ok(0);
    }

    // totalLiquidity = available + borrowed_sf / 2^60
    let total_liquidity = available + borrowed_sf / SCALE_FACTOR;
    let underlying = total_ctokens * total_liquidity / ctoken_supply;

    Ok(underlying as u64)
}
```

If an obligation contains deposits in multiple reserves, this produces a nonsensical “total collateral” value (mixed cTokens treated as if they

belonged to the one reserve). The resulting collateral value is then used for sizing and sanity checks (e.g., proportional repay sizing), which can lead to mis-sized repayments, incorrect health assumptions, and unexpected transaction failures (DoS/UX).

This is practically reachable because Hobba does not strictly enforce a single collateral reserve/mint per `UserState` position. In particular, the collateral mint used to derive the `UserState` PDA is not locally bound to the `supply_token_mint` used for the Kamino deposit CPI (see L5), so multiple deposits into different reserves can be made against the same obligation if the account set is constructed accordingly.

Recommendation

Enforce a single reserve/mint per position bind the deposit inputs (mint, reserve, and PDA source token account) to `user_state.collateral_mint` so a `UserState` position cannot deposit to arbitrary Kamino reserves (see L5).

Fix 1.1

The client fixed the issue by filtering Kamino obligation deposit entries by `deposit_reserve` when summing cTokens and only converting deposits from the provided reserve using that reserve's exchange rate.

Go back to Findings Summary

W4: JupLend prepare_withdraw computes repay sizing from vault_state without refreshing exchange prices

Impact:	Warning	Likelihood:	N/A
Target:	programs/hobba/src/instructions/user/juplend/prepare_withdraw.rs	Type:	Logic error

Description

In the JupLend `prepare_withdraw` flow, Hobba computes the proportional USDC repay amount by reading the JupLend `position` and `vault_state` and converting via exchange prices stored in `vault_state`.

Listing 16. Excerpt from prepare_withdraw

```
/// Compute proportional USDC to repay to maintain current LTV.
/// Reads debt and collateral from position + vault_state with exchange price conversion.
pub fn compute_repay_amount<'info>(
    lender_remaining: &[AccountInfo<'info>],
    collateral_to_withdraw: u64,
    collateral_decimals: u8,
) -> Result<u64> {
    let position = &lender_remaining[0];
    let vault_state = &lender_remaining[1];

    let total_debt = read_juplend_debt_usdc(position, vault_state)?;
    let total_collateral = read_juplend_collateral_native(position, vault_state, collateral_decimals)?;

    let repay = compute_proportional_repay(total_debt, total_collateral, collateral_to_withdraw)?;
    msg!("JupLend LTV maintenance: debt={}, collateral={}, withdraw={}, repay={}",
        total_debt, total_collateral, collateral_to_withdraw, repay);
    Ok(repay)
}
```

Unlike some other JupLend flows, `prepare_withdraw` does not refresh exchange prices on-chain before reading debt/collateral. The freshness is assumed to come from the off-chain transaction builder providing a sufficiently up-to-date `vault_state`.

By contrast, `operator/juplend/harvest_vault` explicitly calls JupLend's `update_exchange_prices` CPI before reading the debt, to avoid under-reporting due to stale exchange prices: This is already implemented in the JupLend harvest path via an `update_exchange_prices` CPI before reading debt.

If `vault_state` exchange prices are stale, `prepare_withdraw` can compute a repay amount that does not correctly maintain the intended LTV, causing unexpected failures or degraded UX (e.g., withdraw attempts reverting or withdrawing less than expected).

Recommendation

Consider refreshing exchange prices inside the user-facing JupLend withdrawal sizing flow (or explicitly document and enforce that the transaction builder must include a fresh `vault_state`, e.g., by requiring an `update_exchange_prices` instruction earlier in the same transaction).

Fix 1.1

The client fixed the issue by refreshing JupLend exchange prices on-chain in the user-facing repay sizing flow (calling `update_exchange_prices` before reading debt/collateral from `position` and `vault_state`).

Go back to Findings Summary

W5: Migration windup trusts user-supplied Kamino flash fee when computing new borrow amount

Impact:	Warning	Likelihood:	N/A
Target:	programs/hobba/src/instructions/operator/migrate/juplend_windup.rs	Type:	Logic error

Description

Both migration windup instructions accept `kamino_flash_fee` as an instruction argument and use it directly when computing the target USDC balance and the amount to borrow from the destination provider:

Listing 17. Excerpt from juplend_windup

```
68 /// `flash_amount` is the principal the trailing flash_repay needs to return
   to Kamino.
69 /// `kamino_flash_fee` is the additional fee Kamino takes on the flash loan
   (0.001%
70 /// = `flash_amount / 100_000` for the main-market USDC reserve, computed
   off-chain).
71 /// We borrow exactly `flash_amount + kamino_flash_fee - pda_usdc_balance`
   from JupLend
72 /// — the minimum needed to fund the flash_repay — so the user's resulting
   JupLend debt
73 /// is just `actual_kamino_debt + kamino_flash_fee` (no operator skim).
74 ///
75 /// remaining_accounts layout:
76 /// [0]    juplend_program
77 /// [1..]  juplend operate accounts (27 fixed + variable branches/ticks)
78 pub fn handler<'info>(
79     ctx: Context<'_, '_, '_, 'info, MigrateJuplendWindup<'info>>,
80     flash_amount: u64,
81     kamino_flash_fee: u64,
82 ) -> Result<()> {
83     require!(flash_amount > 0, HobbaError::ZeroAmount);
84
85     let remaining = ctx.remaining_accounts;
```

```
86      require!(remaining.len() >= 1, HobbaError::InvalidLendingProvider);
87      let juplend_program = &remaining[0];
88      let operate_accounts = &remaining[1..];
89
90      // Collateral to deposit = (balance - 1). JupLend's operate internally
    transfers
91      // (col_amount + 1) tokens from the source ATA to compensate for ceiling-
    division
92      // rounding (cf. user-flow juplend_deposit_borrow.rs which pre-funds the
    PDA with
93      // amount+1). The PDA holds exactly what Kamino returned, so we pass
    amount-1
94      // and JupLend's +1 lands on the actual balance. Costs us 1 lamport of
    dust.
95      let pda_balance = read_token_balance(&ctx.accounts.pda_collateral_ata)?;
96      require!(pda_balance > 1, HobbaError::ZeroAmount);
97      let collateral_amount = pda_balance - 1;
98
99      let user_key = ctx.accounts.user.key();
100      let collateral_mint_key = ctx.accounts.user_state.collateral_mint;
101      let bump = ctx.accounts.user_state.bump;
102      let seeds = &[
103          USER_STATE_SEED,
104          user_key.as_ref(),
105          collateral_mint_key.as_ref(),
106          &[bump],
107      ];
108      let signer_seeds = &[&seeds[..]];
109
110      // Compute new_borrow on-chain so the user pays only (kamino_flash_fee +
    interest_drift),
111      // not the full off-chain buffer. The PDA's USDC ATA holds whatever
    surplus the
112      // unwind step left (= flash_amount - actual_kamino_debt). We need
    pda_usdc to end up
113      // at (flash_amount + kamino_flash_fee) so the trailing bridge to
    operator funds the
114      // flash_repay exactly. Solving: new_borrow = flash_amount +
    kamino_flash_fee - pda_usdc.
115      let pda_usdc_before = read_token_balance(&ctx.accounts.pda_usdc_ata)?;
116      let target_pda_usdc = flash_amount
117          .checked_add(kamino_flash_fee)
118          .ok_or(HobbaError::MathOverflow)?;
119      require!(target_pda_usdc >= pda_usdc_before,
    HobbaError::FlashLoanShortfall);
120      let new_borrow = target_pda_usdc - pda_usdc_before;
```

The same pattern applies in `migrate_kamino_windup`.

If the supplied fee is too low, the trailing Kamino flash repay can fail and revert the transaction.

If the supplied fee is too high, Hobba can borrow extra USDC during migration, transfer the resulting PDA USDC to `operator_usdc_ata` for the flash repay flow, and end up with excess USDC remaining in the operator's token account after the flash repay completes. This increases the user's destination-provider debt beyond what is strictly required to repay the flash loan.

In combination with the missing migration phase/sequence enforcement described in H1, this becomes stronger than a mere misconfiguration issue: an operator can self-fund the unwind leg, skip a real flash-loan sequence entirely, and pocket the full fee component (or any inflated fee component) because Hobba still returns `flash_amount + kamino_flash_fee` to `operator_usdc_ata`.

Exploit scenario

Alice is being migrated by Bob, her assigned operator.

1. Bob self-funds the unwind leg with ordinary USDC rather than using a real Kamino flash borrow.
2. Bob calls the windup instruction with an inflated `kamino_flash_fee`.
3. Hobba computes `new_borrow` from the supplied fee, borrows that larger amount from the destination provider, and bridges the resulting PDA USDC balance back to `operator_usdc_ata`.
4. Because there is no real flash lender to repay, the fee component becomes direct operator profit while Alice is left with excess destination-provider debt.

Recommendation

Derive the flash fee on-chain from Kamino reserve configuration (or validate that the supplied `kamino_flash_fee` matches the on-chain computed value within a tight bound), so the user's post-migration debt cannot be inflated via an arbitrary input.

This should be paired with the sequencing fix from H1 so the "flash repay" bridge cannot be used in transactions that never actually include a matching flash-loan lifecycle.

Fix 1.1

The client fixed the issue by removing the affected migration windup instruction(s).

Go back to Findings Summary

W6: initialize_program is unrestricted and allows the first caller to become admin

Impact:	Warning	Likelihood:	N/A
Target:	programs/hobba/src/instructions/admin/initialize_program.rs	Type:	Logic error

Description

`initialize_program` creates the global `ProgramConfig` PDA and sets `program_config.admin` to the transaction signer (`admin`).

The instruction does not verify that the signer is the deployer, upgrade authority, or any pre-authorized admin key.

Listing 18. Excerpt from initialize_program

```
pub struct InitializeProgram<'info> {
    #[account(mut)]
    pub admin: Signer<'info>,

    #[account(
        init,
        payer = admin,
        space = ProgramConfig::LEN,
        seeds = [PROGRAM_CONFIG_SEED],
        bump,
    )]
    pub program_config: Account<'info, ProgramConfig>,

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

If the program is deployed but `initialize_program` has not yet been executed, any account can call it first and become the configured admin, gaining the ability to perform admin-only actions gated by `program_config.admin`.

Exploit scenario

1. The program is deployed but `initialize_program` has not been called yet.
2. Bob observes the deployment and front-runs the intended initializer by calling `initialize_program` first.
3. Bob becomes `program_config.admin` and can execute admin-gated instructions that rely on `program_config.admin == admin.key()`.

Recommendation

Gate `initialize_program` on the program upgrade authority (via the program data account) or hardcode/derive the expected admin key so that only an authorized entity can perform the one-time initialization.

Acknowledgment 1.1

The client acknowledged the issue.

Fix 1.2

The client fixed the issue by adding a check to verify that the signer is the program upgrade authority.

Go back to Findings Summary

W7: Allowlist enforcement applies to exit actions and can block user repayments and withdrawals

Impact:	Warning	Likelihood:	N/A
Target:	programs/hobba/src/token_utils.rs, programs/hobba/src/instructions/user/prepare_withdraw.rs	Type:	Logic error

Description

When `program_config.allowlist_enabled` is true, Hobba enforces allowlist membership by requiring the allowlist entry PDA to exist:

Listing 19. Excerpt from token_utils

```
/// Validate the allowlist check: if enabled, the entry PDA must exist (have data).
pub fn validate_allowlist(allowlist_enabled: bool, allowlist_entry: &AccountInfo) -> Result<()> {
    if allowlist_enabled {
        require!(!allowlist_entry.data_is_empty(), HobbaError::NotOnAllowlist);
    }
    Ok(())
}
```

This allowlist check is applied to both onboarding actions and exit / risk-reducing actions. For example, `prepare_withdraw` calls `validate_allowlist` before doing any work, and `repay` / `withdraw` follow the same pattern:

As a result, removing a wallet from the allowlist after it has opened a position can prevent the user from repaying debt, preparing a vault unwind, or withdrawing collateral through the normal user flows.

Exploit scenario

Alice is allowlisted and opens a position (deposit + borrow).

1. An admin removes Alice's allowlist entry while keeping `allowlist_enabled = true`.
2. Alice attempts to call `repay` to reduce her debt, but the instruction fails at the allowlist check.
3. Alice attempts to call `prepare_withdraw` and `withdraw` to exit, but those instructions also fail at the allowlist check.
4. Alice's position becomes dependent on admin/operator intervention, even though her requested actions would reduce protocol exposure or return her collateral.

Recommendation

Separate onboarding restrictions from exit permissions.

For example, consider enforcing allowlist checks only on entry actions (`initialize_user`, `deposit`, `borrow`), while allowing exits (`repay`, `prepare_withdraw`, `withdraw`) regardless of allowlist status (or under a separate "exit allowlist" policy).

Acknowledgment 1.1

The client acknowledged the issue.

Go back to Findings Summary

I1: Supported collateral and debt assets are not enforced on-chain

Impact:	Info	Likelihood:	N/A
Target:	programs/hobba/src/instructions/user/initialize_user.rs, programs/hobba/src/instructions/admin/set_user_deposit_limits.rs, programs/hobba/src/instructions/user/deposit.rs	Type:	Data validation

Description

The client stated that Hobba supports only cbBTC collateral and USDC debt. The program does not enforce that product scope on-chain.

`CBBTC_MINT` and `USDC_MINT` are defined in `constants.rs`, but `CBBTC_MINT` is unused. `USDC_MINT` is only used to pin USDC ATAs in `harvest_vault`; `usdc_mint` itself is not constrained on user/operator borrow and repay paths.

User flows take `collateral_mint`, `supply_token_mint`, and `usdc_mint` as unchecked accounts. `set_deposit_limits` and `initialize_user` accept any `collateral_mint` once the admin creates a `DepositConfig` PDA for it. There is no mint allowlist beyond per-mint deposit limits.

A deployment can therefore support additional Kamino/JupLend markets if the integrator passes matching `remaining_accounts`, even though the intended product is cbBTC/USDC only.

Recommendation

If only cbBTC/USDC are in scope, enforce on-chain:

- `collateral_mint == CBBTC_MINT` (and `supply_token_mint` matches it on deposit); and
- `usdc_mint == USDC_MINT` on all USDC transfer/borrow/repay paths.

Alternatively, document that asset support is off-chain/integrator-enforced and remove unused mint constants to avoid implying on-chain enforcement.

Acknowledgment 1.1

The client acknowledged the issue.

Go back to Findings Summary

I2: Several instructions include unused System Program accounts

Impact:	Info	Likelihood:	N/A
Target:	programs/hobba/src/instructions/user/repay.rs	Type:	Code quality

Description

Multiple instructions include a `system_program: Program<'info, System>` account even though the handler does not use it directly.

For example, `repay` declares a `system_program` account but the handler performs only SPL token transfers and lender CPIs without requiring the System Program.

Listing 20. Excerpt from repay

```
47 /// Token program
48 /// CHECK: Token program
49 pub token_program: UncheckedAccount<'info>,
50
51 /// System program
52 pub system_program: Program<'info, System>,
```

This is not a security issue by itself, but it increases account surface area and transaction size, and may confuse integrators about required accounts.

Recommendation

Remove unused `system_program` accounts from instruction interfaces where they are not needed.

If a provider CPI requires the System Program, pass it explicitly in the provider-specific `remaining_accounts` list (or add it as a named account only for those specific instructions).

Fix 1.1

The client fixed the issue by removing the unused `system_program`.

Go back to Findings Summary

I3: deposit does not validate token_program and uses it as CPI target

Impact:	Info	Likelihood:	N/A
Target:	programs/hobba/src/instructions/user/deposit.rs	Type:	Data validation

Description

The `deposit` instruction accepts `token_program` as an `UncheckedAccount` without validating that it is the canonical SPL Token program id.

Listing 21. Excerpt from deposit

```
51 /// cbBTC mint
52 /// CHECK: Validated by token program
53 pub supply_token_mint: UncheckedAccount<'info>,
54
55 /// Token program
56 /// CHECK: Token program
57 pub token_program: UncheckedAccount<'info>,
```

This unvalidated `token_program` is then used in CPIs:

- It is passed to ATA creation (as the token program used by the ATA program) when `pda_source_liquidity` needs to be created.
- It is used as the CPI target program id for `transfer` when moving tokens from the user to the PDA.

Because `pda_source_liquidity` is created only once, subsequent deposits do not invoke the ATA program at all, so there is no indirect validation that the supplied `token_program` is correct.

This is currently limited to user-signed flows: a malicious/compromised transaction builder could substitute a different executable program id as `token_program`, causing Hobba to invoke an unintended program during

deposit.

Recommendation

Constrain `token_program` to the expected SPL Token program id (and/or explicitly support Token-2022 by validating against a known allowlist and using token-interface style validation).

Fix 1.1

The client fixed the issue by changing `token_program` to `Program<Token>` in `deposit`, which pins the CPI target to the canonical SPL Token program id.

Go back to Findings Summary

Report Revision 1.1

Revision Team

Member's Name	Position
Andrej Lukačovič	Lead Auditor
Matej Hyčko	Auditor
Artem Delikatnyi	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.

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

H2: Kamino CPIs do not validate the Kamino program id, enabling arbitrary CPI with UserState PDA signer

High severity issue

Impact:	High	Likelihood:	Medium
Target:	programs/hobba/src/instructions/user/kamino/*.rs, programs/hobba/src/instructions/operator/kamino/*.rs, programs/hobba/src/kamino_cpi.rs	Type:	Data validation

Description

Multiple Kamino-routed instructions accept the Kamino program account via `remaining_accounts` and then use it as the CPI target program id in `CpiContext::new_with_signer(…)`, with the `UserState` PDA acting as the CPI signer.

The program does not validate that the supplied `kamino_program` account equals the canonical Kamino KLend program id (`KAMINO_LENDING_PROGRAM_ID`). As a result, a caller can substitute an arbitrary executable program as `kamino_program`.

Because Hobba signs these CPIs using `UserState` PDA seeds and passes PDA-owned token accounts (e.g., PDA collateral/USDC ATAs) as writable accounts, a substituted program can use the PDA signature to execute unintended token operations (e.g., transfer PDA-held tokens to an attacker-controlled destination), or otherwise violate protocol invariants.

This issue is particularly dangerous because the outer instruction still

appears as a legitimate call into the Hobba program (correct Hobba program id and instruction discriminator). The malicious behavior is hidden behind the CPI boundary: the inner CPI target program id is attacker-controlled via `remaining_accounts`.

In practice, the injection happens because handlers extract the Kamino program account directly from a fixed index in `lender_remaining` and forward it into CPI wrappers without a key check. For example, the user Kamino withdraw path takes `kamino_program` from `lender_remaining[13]` and passes it into `invoke_withdraw_v2(…)`:

Listing 22. Excerpt from withdraw

```
let farms_program = &lender_remaining[12];
let kamino_program = &lender_remaining[13];
let instruction_sysvar = &lender_remaining[14];

// Kamino WithdrawV2 takes cToken (collateral) amount, not underlying.
// Read PDA balance before/after to determine actual underlying received.
let balance_before = read_token_balance(&accounts.pda_destination_ata)?;

invoke_withdraw_v2(
    user_state.to_account_info(),
    obligation.clone(),
    lending_market.clone(),
    lending_market_authority.clone(),
    withdraw_reserve.clone(),
    reserve_liquidity_mint.clone(),
    reserve_source_collateral.clone(),
    reserve_collateral_mint.clone(),
    reserve_liquidity_supply.clone(),
    accounts.pda_destination_ata.to_account_info(),
    placeholder_user_destination_collateral.clone(),
    collateral_token_program.clone(),
    accounts.token_program.to_account_info(),
    instruction_sysvar.clone(),
    obligation_farm_user_state.clone(),
    reserve_farm_state.clone(),
    farms_program.clone(),
    kamino_program.clone(),
    collateral_amount,
```

```
74     signer_seeds,
```

The wrapper then performs the CPI using the supplied `kamino_program` as the `CpiContext` program:

Listing 23. Excerpt from kamino_cpi

```
179 kamino_lending::cpi::withdraw_obligation_collateral_and_redeem_reserve_colla
    teral_v2(
180     CpiContext::new_with_signer(kamino_program, cpi_accounts, signer_seeds),
181     collateral_amount,
182 )
```

In the Anchor-generated CPI helper, the destination program id of the constructed instruction is taken from the CPI context (i.e., the `kamino_program` account above):

Listing 24. Excerpt from Anchor-generated CPI helper (illustrative)

```
1 anchor_lang::solana_program::instruction::Instruction {
2     program_id: ctx.program.key(),
3     accounts,
4     data,
5 }
```

Once Hobba performs `invoke_signed` (via `CpiContext::new_with_signer`) with `UserState` seeds, the runtime marks the `UserState` account as a signer for the invoked program. A substituted program can then perform arbitrary CPIs using that already-granted signer privilege (it does not need to know or re-derive the PDA seeds).

Exploit scenario

Alice is a Hobba user (victim). Bob is an attacker who controls Alice's operator role.

1. Bob calls an operator-only Kamino-routed instruction (e.g., `operator_repay`

/ `harvest_vault`) for Alice's `UserState` and supplies a malicious executable program account in place of the Kamino KLend program in `remaining_accounts`.

2. Hobba invokes the supplied program via `CpiContext::new_with_signer(…)`, providing the `UserState` PDA signature and writable PDA token accounts in the CPI account list.
3. The malicious program uses the PDA signature to transfer PDA-held tokens (e.g., USDC in `pda_usdc_ata`) to Bob's token account.
4. Hobba continues execution as if a Kamino CPI had been performed successfully, but Alice's funds have been transferred to Bob.

Alternatively, this can be exploited via a forged "legit-looking" Hobba transaction where the malicious behavior is isolated to `remaining_accounts`:

1. Bob tricks Alice into signing a transaction that calls a legitimate Hobba instruction and shows expected top-level accounts.
2. The transaction silently substitutes the Kamino KLend program account inside `remaining_accounts` with an attacker-controlled executable program (and supplies any other attacker-chosen remaining accounts to satisfy the CPI interface).
3. On-chain, the instruction still looks like a normal Hobba call (correct program id and instruction discriminator), but Hobba will CPI into Bob's program because it trusts `remaining_accounts`.
4. The attacker program can return success while misdirecting funds; e.g., on `deposit`, it can redirect the would-be Kamino deposit to an attacker destination via the attacker-controlled `remaining_accounts`.

Recommendation

Validate all of the Kamino CPI program ids before performing any Kamino CPI. Apply the same pattern to any other program-id accounts supplied through

`remaining_accounts` (e.g., farms program, token programs, kVault program).

Fix 1.2

The client fixed the issue by introducing `require_kamino_program` in `program_guards.rs` and calling it at the start of every Kamino CPI wrapper in `kamino_cpi.rs` before any signed CPI is executed. A substituted `kamino_program` in `remaining_accounts` now fails with `InvalidKaminoProgram`.

Go back to Findings Summary

W8: Operator JupLend harvest does not pin the JupLend program id for update_exchange_prices CPI

Impact:	Warning	Likelihood:	N/A
Target:	programs/hobba/src/instructions/operator/juplend/harvest_vault.rs, programs/hobba/src/juplend_cpi.rs	Type:	Data validation

Description

The operator-only JupLend harvest path accepts the JupLend program account via `remaining_accounts` and then uses it as the destination program id for the `update_exchange_prices` CPI.

Unlike `operate`, which builds an instruction with `program_id = JUPLEND_PROGRAM_ID`, the `update_exchange_prices` wrapper uses `CpiContext::new(juplend_program, …)`. If `juplend_program` is not pinned to `JUPLEND_PROGRAM_ID` at the call site, an attacker-controlled operator can substitute an arbitrary executable program.

Excerpt showing `juplend_program` taken from `lender_remaining`:

Listing 25. Excerpt from harvest_vault

```
22 let pda_source_liquidity = &lender_remaining[0];
23 let juplend_program = &lender_remaining[1];
24 let juplend_operate_remaining = &lender_remaining[2..];
```

Excerpt showing the unpinned `juplend_program` forwarded into `invoke_update_exchange_prices(…)`:

Listing 26. Excerpt from harvest_vault

```
invoke_update_exchange_prices(
    juplend_program.clone(),
    vault_state_account.clone(),
    vault_config_account.clone(),
    supply_reserves.clone(),
    borrow_reserves.clone(),
    vault_id,
)?;
```

Excerpt showing the CPI uses the passed program account:

Listing 27. Excerpt from juplend_cpi

```
pub fn invoke_update_exchange_prices<'info>(
    juplend_program: AccountInfo<'info>,
    vault_state: AccountInfo<'info>,
    vault_config: AccountInfo<'info>,
    supply_token_reserves_liquidity: AccountInfo<'info>,
    borrow_token_reserves_liquidity: AccountInfo<'info>,
    vault_id: u16,
) -> Result<()> {
    let cpi_accounts = juplend::cpi::accounts::UpdateExchangePrices {
        vault_state,
        vault_config,
        supply_token_reserves_liquidity,
        borrow_token_reserves_liquidity,
    };
    juplend::cpi::update_exchange_prices(
        CpiContext::new(juplend_program, cpi_accounts),
        vault_id,
    )
```

Recommendation

Always validate the JupLend program id before performing any JupLend CPI. Apply the same pattern to any other program-id accounts supplied through `remaining_accounts` (e.g., farms program, token programs, kVault program).

Fix 1.2

The client fixed the issue by introducing `require_juplend_program` in `program_guards.rs` and calling it inside `invoke_update_exchange_prices` (and the other JupLend CPI wrappers) before the CPI runs. A substituted `juplend_program` in `remaining_accounts` now fails with `InvalidJuplendProgram`.

Go back to Findings Summary

Report Revision 1.2

Revision Team

Member's Name	Position
Andrej Lukačovič	Lead Auditor
Matej Hyčko	Auditor
Artem Delikatnyi	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 1.2.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Audit Report | Hobba: Prime broker for onchain loans on Solana, June 15, 2026.

Appendix B: 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.

B.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: Migration windup/unwind lack on-chain phase gating, enabling standalone operator-funded migrations and unearned fee extraction	**High**	1.0	Trident Arena, Auditor
M2: User-facing token accounts are not bound to canonical ATAs, enabling destination substitution in user-signed flows	Medium	1.0	Trident Arena, Auditor
M3: JupLend borrow skips lender borrow after partial vault withdrawal	Medium	1.0	Trident Arena, Auditor
M4: claim_vault_rewards passes UserState as a writable CPI signer while the outer account is read-only	Medium	1.0	Trident Arena, Auditor

Finding title	Severity	Reported	Discoverer
L2: claim_vault_rewards does not validate the Associated Token Program before invoking ATA creation CPI	Low	1.0	Trident Arena, Auditor
L3: set_treasury allows misconfiguration that can block harvest_vault fee transfers	Low	1.0	Trident Arena, Auditor
L4: migrate_juplend_unwind does not verify collateral is fully withdrawn	Low	1.0	Trident Arena, Auditor
L5: deposit does not bind supply_token_mint to user_state collateral mint	Low	1.0	Trident Arena, Auditor
L6: Vault flows have inconsistent min-out / slippage enforcement (Perena)	Low	1.0	Trident Arena, Auditor
L7: JupLend flows do not bind operate account list to UserState fields, allowing state/CPI desynchronization	Low	1.0	Trident Arena, Auditor
L8: switch_vault accepts a non-canonical source share account and can strand old vault shares	Low	1.0	Trident Arena, Auditor
L9: Perena vault CPIs do not bind vault_remaining to the named vault and mints	Low	1.0	Trident Arena, Auditor

Finding title	Severity	Reported	Discoverer
W4: JupLend prepare_withdraw computes repay sizing from vault_state without refreshing exchange prices	Warning	1.0	Trident Arena, Auditor
W5: Migration windup trusts user-supplied Kamino flash fee when computing new borrow amount	Warning	1.0	Trident Arena, Auditor
W6: initialize_program is unrestricted and allows the first caller to become admin	Warning	1.0	Trident Arena, Auditor
W7: Allowlist enforcement applies to exit actions and can block user repayments and withdrawals	Warning	1.0	Trident Arena, Auditor
I2: Several instructions include unused System Program accounts	Info	1.0	Trident Arena, Auditor
I3: deposit does not validate token_program and uses it as CPI target	Info	1.0	Trident Arena, Auditor
H2: Kamino CPIs do not validate the Kamino program id, enabling arbitrary CPI with UserState PDA signer	**High**	1.1	Trident Arena, Auditor
W8: Operator JupLend harvest does not pin the JupLend program id for update_exchange_prices CPI	Warning	1.1	Trident Arena, Auditor

Table 4. 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