Florianópolis, SC

November, 2025

Audit Threat Analysis and Invariant Specification Report - KipuBankV3

Audits Team

Luis Felipe Fabiane

Document version control

Version	Date	Description
v1.0	25/11/2025	First audit version.

Introduction

This document presents the **Threat Analysis Report** for the **KipuBankV3** protocol, as requested for Deliverable 5. The objective of this document is to provide a comprehensive security assessment, identifying vulnerabilities, analyzing code maturity, and detailing the risks associated with actors and their privileges in the protocol's ecosystem.Fundamental Concepts

Core concepts

KipuBankV3 is a decentralized finance (DeFi) protocol designed to resemble an internal "bank" where users deposit ETH or any token supported by Uniswap V4, the value is always converted to USDC, and the user then has an internal balance denominated in USDC. The contract also allows withdrawal in USDC or ETH, with withdrawal limits based on USD via the Chainlink price feed. It operates through a set of guidelines that define the business logic, transaction rules, and asset management.

Security is paramount, given the immutable nature of contracts and the financial value they manage. Threat analysis is the systematic process of identifying and assessing potential weaknesses (threats and vulnerabilities) that an attacker could exploit to harm the protocol, such as fund theft, denial of service, or data manipulation.

The security structure of KipuBankV3 is supported by a specific Trust Model. This model details the hierarchy of trust between users (e.g., depositors), administrators, and external mechanisms (e.g., price oracles). The protocol's security depends not only on the correctness of the code itself (which is verified by critical Invariants) but also on minimizing centralized privileges and the robustness of governance and update processes.

Sumário

1. Protocol Overview

1.1 Purpose of the Protocol

KipuBankV3 acts as a custodial "bank-like" smart contract that accepts deposits in ETH or any ERC20 token supported by Uniswap v4 routing. All deposits are automatically swapped into **USDC**, and the user receives an **internal USDC balance** stored in the contract's accounting.

Withdrawals can occur in:

- **USDC**, directly
- **ETH**, via a reverse swap (USDC → ETH) routed through the UniversalRouter

ETH withdrawals are subject to **USD-denominated withdrawal limits** enforced via Chainlink price feeds.

1.2 Core Components

Component	Description
UniversalRouter (Uniswap v4)	Handles swaps during deposits and ETH withdrawals
USDC (immutable)	Denomination currency and internal accounting unit
Chainlink Price Feeds	Ensure ETH withdrawal caps and conversion correctness
AccessControl (ADMIN_ROLE)	Allows protocol governance and privileged operations
ReentrancyGuard	Prevents reentrancy attacks in critical functions
Bank Cap (bankCapUsdc)	Maximum total USDC held by the protocol
nativePerTxCapWei	Optional hard cap for ETH withdrawals

1.3 High-Level Process Flow

Deposits

1. User deposits ETH or ERC20 token.
2. Token is swapped into USDC via UniversalRouter.
3. USDC received is credited to the internal user balance.
4. Total bank liquidity increases.

Withdrawals

1. User withdraws USDC
   OR
   requests ETH → which requires conversion USDC → ETH.
2. Internal USDC balance is deducted.
3. Swap executed (if ETH).
4. ETH or USDC is transferred to the user.

2. Protocol Maturity Assessment

2.1 Strengths

- Effective use of **CEI pattern** (Checks → Effects → Interactions).
- Good use of **ReentrancyGuard**.
- Comprehensive **price feed validation**:
    - Staleness check
    - Positive answer validation
    - Round completion validation
- Secure and minimal use of token approvals (always reset to zero).
- Clear access control boundaries and event emission.

2.2 Weaknesses / Missing Maturity Steps

A. Testing Coverage (Missing)

The contract needs:

- Unit tests for:
    - Deposit flows (ETH, USDC, ERC20)
    - Withdrawal flows (USDC + ETH)
    - Bank cap enforcement
    - Price feed staleness
    - Failure scenarios (bad router instructions)

- Integration tests using:
    - Mainnet forks
    - Mocked Uniswap router
    - Mocked Chainlink oracles

Recommended coverage: ≥ 95%.

B. Testing Methods Not Specified

The project must define:

- **Property-based tests** (via Foundry `forge` fuzzing)
- **Invariant testing** (slippage, balances, accounting)
- **Differential testing** (mock vs. mainnet interactions)
- **Symbolic execution** (Mythril)
- **Static analysis** (Slither)

C. Documentation Gaps

Missing:

- Full protocol architecture diagram
- Router instruction specifications
- Safety assumptions
- Administrative model rationale
- Risk disclosures for custodial responsibilities

D. Permission Model Risks

The **ADMIN_ROLE** has extremely powerful abilities:

- Can arbitrarily modify user balances (`changeUserBalance`)
- Can change price oracles
- Can change withdrawal limits
- Can increase bank cap
- Can create or destroy effective internal liquidity

This is a custodial model — but it needs explicit documentation and potentially **multisig governance**.

E. Operational Maturity Missing

To achieve production readiness, the protocol still needs:

- On-chain monitoring + alerting
- Operational runbooks
- Slippage protection in router instructions (currently fully user-defined)
- Pause mechanism for crises (`Pausable` recommended)
- Emergency withdrawal only for admins (if router fails)

3. Threat Model and Attack Surface

Threat 1 — Business Logic Manipulation via `changeUserBalance`

Surface

`ADMIN_ROLE` can **arbitrarily change any user's balance**, including increasing it without corresponding USDC reserves (until bank cap is reached).

Risk

- Rogue admin can mint internal USDC from thin air.
- Bank's internal accounting can desynchronize from actual USDC holdings.
- Users may expect redeemability that is no longer economically possible.

Impact

- Insolvency
- Loss of funds for all users
- Total protocol failure

Mitigation

- Restrict `changeUserBalance` to only decrease balances
- OR require:
    - multisig
    - timelock
    - event monitoring

Threat 2 — Malicious Router Instructions Provided by Users

Surface

Users pass arbitrary `routerInstructions` directly to Uniswap's UniversalRouter.

Risks

- Instructions may:
    - attempt to steal approvals (unlikely but router may support calls to arbitrary contracts)
    - cause the router to execute unexpected callback logic
    - cause the contract to receive unexpected tokens / ETH
    - revert the entire transaction (DoS)

Impact

- Unexpected ETH stuck in contract
- Inability for users to withdraw (router always reverts)
- Loss of accounting integrity

Mitigation

- Whitelist only specific instruction formats
- Add an internal DSL or input validation layer
- Limit router operations to a known action selector

Threat 3 — Oracle / Price Feed Manipulation or Failure

Surface

The protocol uses Chainlink price feeds to:

- enforce USD-denominated withdrawal limits
- calculate required USDC for ETH withdrawals

Risks

- Stale or zero price halts ETH withdrawals entirely
- Oracle replacement by admin (malicious oracle configured)
- Flash loaning ETH price if fallback or new feed is improperly configured

Impact

- DoS on ETH withdrawals
- Users able to drain ETH cheaply (if price too low)
- Users prevented from withdrawing ETH (if price too high)

Mitigation

- Lock oracle address after initialization (no setter)
- Require multisig governance for oracle changes
- Monitor oracle freshness externally
- Add secondary oracle fallback

4. Protocol Invariants

Invariant 1 — Total User Balances Must Equal Total Bank Assets

Definition:

```
SUM(usdcBalances) == totalBankUsdc
```

This invariance must always hold post-transaction.

Violation Impact

- Internal accounting no longer reflects actual collateral
- Insolvency
- Users can no longer redeem holdings
- Possible hidden inflation of balances

Validation Recommendations

- Add invariant tests in Foundry:
    - After every deposit/withdraw/fuzz path
- Log every assignment to totalBankUsdc and assert non-corruption
- Consider removing admin functions that break this invariant

Invariant 2 — ETH Withdrawals Must Never Exceed USD Price Limits

Definition:

```
requestedWei <= maxWeiFromUsdLimit(usdWithdrawLimit8)
```

This must always hold.

Violation Impact

- User can bypass withdrawal limits
- Possible draining of protocol capital
- Allows large-scale arbitrage extraction

Validation Recommendations

- Fuzz tests with random ETH prices
- Unit tests using fixed staleness and price values
- Reject or pause when price feed is compromised

Invariant 3 — Contract Cannot Hold Residual Approvals for Router

Definition:

`ERC20.allowance(this, universalRouter) == 0 after operation`

Violation Impact

- Router (or malicious router replacement) can spend tokens unexpectedly
- Approvals can be used in future swaps outside user control

Validation Recommendations

- Add test that every deposit/withdraw resets allowance to 0
- Add invariant property tests using Foundry `invariant_*`

5. Impact of Invariant Violations

Invariant	Impact
#1 Accounting correctness	Insolvency, loss of funds, unredeemable balances
#2 Withdrawal limit enforcement	Exploit allowing unlimited ETH withdrawals vs. USDC backing
#3 Allowance safety	Router or attacker drains USDC instantly

6. Recommendations for Hardening

High Priority

- Replace `changeUserBalance` with controlled, auditable mechanisms
- Introduce **Pausable** to respond to oracle failures
- Add **multisig** for ADMIN_ROLE
- Restrict oracle updates or freeze after deployment
- Add slippage protections (max slippage %)

Medium Priority

- Add invariant tests and fuzz testing
- Add router instruction validation
- Add emergency recovery of stuck ETH

Low Priority

- Improve documentation
- Provide reference router instructions in docs

7. Conclusion & Next Steps

KipuBankV3 is structurally sound but **not yet production-ready** due to governance centralization, missing testing, and operational controls.

To reach full protocol maturity:

Required Before Mainnet

1. Full test suite with invariants, fuzzing, and integration tests
2. Governance upgrade to multisig
3. Remove or strictly restrict dangerous admin functions
4. Add configurable emergency pause
5. Add monitoring (oraclize, price-feed checks)
6. Document router restrictions and safety assumptions