Threat Analysis & Invariant Specification Report

1. Protocol Overview

KipuBankV3 is a DeFi “bank” that accepts multiple assets, enforces USD-denominated risk limits, and normalizes arbitrary tokens into USDC using Uniswap V4. It is designed as a custodial protocol that tracks user balances per asset and enforces bank-wide and per-transaction risk constraints.

1.1 Core Concepts and State

Per-user balances:

- **mapping(address => mapping(address => uint256)) public balances;**

  Tracks how many units of each token a user has deposited (including ETH, represented by a special ETH_ADDRESS constant).

Global risk parameters (USD-denominated, 8 decimals):

- **bankCapInUsd:** Maximum total USD value the bank is allowed to hold.
- **withdrawalLimitInUsd:** Maximum USD value a user can withdraw in a single transaction.
- **totalBankValueInUsd:** Aggregate USD value of all assets held by the bank (intended to be kept in sync with user balances via oracle pricing).

Price oracles:

- **tokenPriceFeeds[token] -> priceFeed**: Chainlink **AggregatorV3Interface** for each supported asset.
- **_getPriceUsd8(token)**: Fetches and normalizes oracle prices to 8 decimals.
- **_getValueInUsd(token, amount):** Converts a token amount (using its decimals) to a USD value with 8 decimals.

Special addresses and external components:

- **ETH_ADDRESS**: Sentinel address representing native ETH.
- **WETH**: Wrapped ETH contract used when swapping ETH via Uniswap.
- **USDC**: ERC-20 token used as the canonical internal settlement asset for depositArbitraryToken.
- **IUniversalRouter universalRouter:** Uniswap V4 Universal Router used for swaps.
- **IPermit2 permit2** and **poolManager**: Present in the constructor but not yet used directly in the core flows.

Uniswap V4 pool configuration

- **tokenToUsdcPools[token] -> PoolKey**: Mapping from arbitrary token to Uniswap V4 pool parameters (currencies, fee, tick spacing, hooks) to route **token → USDC** swaps.

1.2 Roles and Access Control

KipuBankV3 uses **AccessControl** to separate concerns and limit privileged operations. All these roles are initially granted to the deployer in the constructor.:

- **DEFAULT_ADMIN_ROLE**: Assigned to the deployer. Can grant and revoke other roles.
- **OPERATIONS_MANAGER_ROLE:** Controls global risk parameters.
  - **setBankCapInUsd(uint256 newCap)**
  - **setWithdrawalLimitInUsd(uint256 newLimit).**
- **ASSET_MANAGER_ROLE:**
  - **addToken(address token, address priceFeed)** -> Registers supported tokens and their Chainlink price feeds.
  - **addTokenPool(address token, PoolKey calldata poolKey)** -> Configures Uniswap V4 pools for token → USDC swaps.
- **FUNDS_RECOVERY_ROLE**
  - **recoverBalance(token, user, newBalance)**: Can arbitrarily adjust per-user balances and correspondingly update **totalBankValueInUsd.** This is a powerful “manual correction” mechanism intended for incident recovery.

1.3 User Flows

Direct Deposit / Withdraw (deposit and withdraw)

This path is a “plain” bank-like custody flow without any automated swapping.

- **Deposit: deposit(token, amount) (payable)**
  For **ETH**:
  - Requires **_tokenAddress == ETH_ADDRESS** and **msg.value == amount.**
  - Computes **depositValueInUsd = _getValueInUsd(ETH_ADDRESS, amount).**

  For **ERC-20 tokens**:

  - Requires no ETH sent **(msg.value == 0).**
  - Calls **IERC20(token).transferFrom(msg.sender,address(this), amount).**
    - Computes **depositValueInUsd = _getValueInUsd(token, amount).**

  In both cases:

  - Checks totalBankValueInUsd + depositValueInUsd <= bankCapInUsd.
  - Increments balances[token][msg.sender] and totalBankValueInUsd.

- Emits Deposit(user, token, amount).

- **Withdraw:** **withdraw(token, amount)**
  - Validates amount > 0 and that **balances[token][msg.sender] >= amount.**
  - Computes **amountInUsd = _getValueInUsd(token, amount)**.
  - Enforces **amountInUsd <= withdrawalLimitInUsd**.
  - Decrements balances[token][msg.sender] and totalBankValueInUsd.
  - For ETH: sends via **call{value: amount}("")**.
  - For ERC-20: calls **IERC20(token).transfer(msg.sender, amount).**
  - Emits **Withdrawal(user, token, amount).**

Arbitrary Token Deposit via Uniswap (**depositArbitraryToken**)

This path is the primary way to accept "any" Uniswap V4-supported token while keeping the bank's internal liability side denominated in USDC.

- **depositArbitraryToken(token, amount, minUsdcOut)** (payable, nonReentrant)

  token == USDC:

  - Directly calls **safeTransferFrom(msg.sender,address(this), amount).**
  - Credits **balances[USDC][msg.sender] += amount.**

  token == ETH_ADDRESS:

  - Requires **msg.value == amount**.
  - Wraps ETH → WETH via **IWETH(WETH).deposit{value: amount}().**
  - Swaps WETH → USDC via **_swapExactInputSingle(WETH, amount, minUsdcOut)**.

  token is any other ERC-20:

  - Requires **msg.value == 0**.
  - Transfers the token from the user to the bank.
  - Swaps token → USDC via **_swapExactInputSingle(token, amount, minUsdcOut)**.

  After swap:

  - Computes **depositValueInUsd = _getValueInUsd(USDC, usdcReceived)**.
  - Enforces bank cap: **totalBankValueInUsd + depositValueInUsd <= bankCapInUsd**.
  - Credits **balances[USDC][msg.sender] += usdcReceived** and updates **totalBankValueInUsd**.
  - Emits **Deposit(user, USDC, usdcReceived)** and, for non-USDC flows, **TokenSwapped**.

- **_swapExactInputSingle(tokenIn, amountIn, minAmountOut):**
    - Reads **PoolKey** from **tokenToUsdcPools[tokenIn]**.
    - Reverts with **PoolNotConfigured** if the pool is not set.
    - Uses **forceApprove** to approve **universalRouter**.
    - Encodes a single-hop path **tokenIn → USDC**.
    - Executes a V4 swap through **universalRouter.execute**.
    - Computes the actual USDC received by comparing USDC balance before/after.
    - Enforces slippage: **amountOut >= minAmountOut** or reverts with **SlippageExceeded**.
    - Resets the allowance back to zero.

2. Protocol Maturity Assessment

2.1 Design Maturity

KipuBankV3 exhibits a clear and modular design:

- **Access control** is explicitly defined via **AccessControl**, with separate roles for operations, asset management, and funds recovery.
- The bank enforces **USD-denominated risk limits** (**bankCapInUsd**, **withdrawalLimitInUsd**) using **Chainlink** price feeds.
- It integrates with **Uniswap V4** through a dedicated routing path (**depositArbitraryToken**) while still supporting traditional deposit/withdraw flows.
- It uses **ReentrancyGuard** for its key external entrypoints, and **SafeERC20** for safe token transfers in the Uniswap-related path.

However, some aspects are still at prototype level from a production standpoint:

- **permit2** and **poolManager** are currently unused, which suggests either incomplete integration or leftover scaffolding.
- **recoverBalance** is a powerful administrative function that can arbitrarily rewrite user balances and total bank value without on-chain proofs or constraints beyond role checks.
- **totalBankValueInUsd** is maintained incrementally and can become inconsistent or even lead to DoS scenarios when prices move significantly between deposit and withdrawal.

2.2 Testing Maturity

The current test suite (**KipuBankV3.t.sol**) is implemented with **Foundry** and uses mock contracts for external dependencies:

Mocks & setup:

- **MockERC20** for USDC, DAI, and WETH.

- **MockChainlinkAggregator** for USDC/USD, DAI/USD, WETH/USD price feeds.
- **MockUniversalRouter** for simulating Uniswap V4 swaps and configurable exchange rates.

What is covered today:

Constructor and configuration

- Tests ensure that the constructor correctly initializes the router, USDC, bank cap, and withdrawal limit.
- Reverts are tested when critical addresses (router or USDC) are zero.

Access control for pool management

- **addTokenPool** succeeds for authorized callers and emits **PoolKeyAdded**.
- Unauthorized attempts to call **addTokenPool** revert, confirming role enforcement.

Deposits via depositArbitraryToken

- Direct USDC deposits (no swap) are tested end-to-end, including event emission and user balance updates.
- Arbitrary token deposits with swap (e.g., DAI → USDC) are tested with:
  - Configured pool parameters.
  - Mocked exchange rates.
  - Assertions on received USDC amount and **TokenSwapped** event.
- Slippage protection is verified by setting an unfavorable rate and ensuring the function reverts when **minUsdcOut** is not satisfied.

Risk controls (bank cap)

- Depositing close to the configured cap and then attempting to exceed it, with the second deposit correctly reverting.

Multi-user scenarios and withdrawals

- Multiple users depositing USDC independently, with correct per-user balances.
- A full USDC deposit–then–withdraw flow (**depositArbitraryToken** followed by **withdraw**) verifying both internal accounting and external token balances.

Error handling

- Reverts when a pool is not configured for a token (**PoolNotConfigured** on WETH deposit).
- Reverts on zero-amount deposits (**InvalidAmount**)

Current gaps in test coverage: From a pre-audit perspective, significant areas remain under-tested:

- The “legacy” **deposit function** (for ETH and ERC-20 without routing) is not explicitly exercised.
- **ETH flows** using **ETH_ADDRESS** and **msg.value** are untested (both for **deposit** and **depositArbitraryToken**).
- **Governance/operations functions** have no dedicated tests:
    - **setBankCapInUsd**
    - **setWithdrawalLimitInUsd**
    - **addToken** (price feed registration)
    - **recoverBalance** (including its impact on **totalBankValueInUsd** and per-user balances).
- **Error paths in withdraw** are only partially covered:
    - There are no explicit tests for **InsufficientBalance** and **WithdrawalAmountExceedsUsdLimit**.
- **Invariants and price dynamics**:
    - No tests simulate price changes between deposit and withdrawal to validate the stability and correctness of **totalBankValueInUsd**.
    - Solvency invariants (sum of user balances per token vs. actual token balance of the contract) are not currently asserted.
- **Advanced testing techniques**:
    - No fuzzing/property-based tests (e.g., via Echidna or Foundry fuzz tests).

Assessment

The existing test suite represents a **solid first layer** of unit tests for the main happy paths and selected failure cases in the Uniswap deposit flow and bank cap logic. However, to reach an audit-ready level of maturity, KipuBankV3 still needs:

- Broader functional coverage (all entrypoints and roles).
- Invariant-based and fuzz testing.
- Integration tests against real infrastructure (via forking).
- Explicit tests that target the main invariants and threat scenarios identified in the threat analysis (solvency, withdrawal limits, **totalBankValueInUsd** behavior, and privileged operations).

2.3 Documentation and Invariants

- The contract includes **NatSpec comments** and custom errors, which help clarify intent at the code level.

However, there is currently no separate protocol specification document describing formal invariants and threat models. These aspects are made explicit later in this report (Section 4 – Invariant Specification) but are not yet encoded in the codebase or test suite.

3. Attack Vectors and Threat Model

This section combines a concise threat model with concrete attack scenarios for KipuBankV3. The focus is on realistic ways the protocol can be broken, including business-logic mistakes, abuse of assumptions, economic strategies, and access-control issues.

3.1 Context: Assets, Actors, Assumptions

Key assets

- User funds held by the contract:
  - ETH, ERC-20 tokens, and especially USDC held after swaps.
- Internal accounting and risk parameters:
  - **balances[token][user]**
  - **totalBankValueInUsd**, **bankCapInUsd**, **withdrawalLimitInUsd**
- Configuration:
  - **tokenPriceFeeds[token]** (Chainlink feeds)
  - **tokenToUsdcPools[token]** (Uniswap V4 pools)
  - Access-control roles and their assigned addresses.

Actors

- **Unprivileged users**: Can call **deposit**, **withdraw**, **depositArbitraryToken**.
- **Privileged roles**:
  - **OPERATIONS_MANAGER_ROLE** – sets caps and limits.
  - **ASSET_MANAGER_ROLE** – manages price feeds and pools.
  - **FUNDS_RECOVERY_ROLE** – can arbitrarily change user balances.
- **External dependencies**:
  - Chainlink price feeds, Uniswap V4 Universal Router, WETH.
- **MEV / network adversaries**:
  - Can front-run or back-run transactions, manipulate on-chain prices, etc.

Assumptions

- Ethereum consensus and the EVM are honest.
- Chainlink and Uniswap behave according to spec (unless misconfigured).
- Accepted tokens behave “ERC-20-like” (no severe rebasing/fee surprises).
- Admin keys are intended to be honest, but their compromise is explicitly considered as a threat.

3.2 Attack Vector 1 – Business-Logic DoS via totalBankValueInUsd and Price Changes

Category: Smart contract business-logic error + misuse of protocol assumptions (price stability).

Relevant code paths:

- **deposit(token, amount)**
- **withdraw(token, amount)**
- **_getValueInUsd(token, amount)**
- **totalBankValueInUsd** updates

Description

The protocol tracks aggregate value in USD using a mutable **totalBankValueInUsd**:

- On deposit: **totalBankValueInUsd += _getValueInUsd(token, amount);** (using *current* price from the oracle).
- On withdraw: **totalBankValueInUsd -= _getValueInUsd(token, amount);** (again, using *current* price from the oracle).

This implicitly assumes that using the **latest** price for both operations is safe and will keep **totalBankValueInUsd** consistent. In reality, this creates a time-of-price-change issue.

Concrete scenario

1. User deposits 100 units of token X when its price is $1:
    - **_getValueInUsd(X, 100) = 100**
    - **totalBankValueInUsd** goes from 0 -> 100.

2. Later, the oracle price for X increases to $2 (for legitimate or manipulated reasons).

3. The same user now tries to withdraw 100 units of X:
    - **_getValueInUsd**(X, **100**) is now 200.
    - The contract attempts: **totalBankValueInUsd = totalBankValueInUsd - 200** but **totalBankValueInUsd** is only 100 -> underflow revert.

Result: **withdrawal reverts**, even though the user is only withdrawing exactly what they deposited and the bank holds enough tokens.

Impact

- **Denial of Service (DoS)**: users may be completely unable to withdraw after price movements, especially upward moves.
- **Global lock-up risk**: if **totalBankValueInUsd** becomes inconsistent or too small relative to current prices, *all* withdrawals of a given asset might revert.
- **Indirect attack vector via oracle manipulation**:
    - An attacker who can temporarily manipulate the oracle price upwards can cause withdrawals to revert for honest users (griefing).
    - Even if the attacker cannot steal funds, they can freeze withdrawals around a given asset.

The invariant the code seems to want is:

"**totalBankValueInUsd** reflects the sum of all deposits minus withdrawals, valued in USD."

However, by using the **current price** on both sides of the lifecycle, the protocol is effectively tracking **mark-to-market value**, but subtracting that from an **historical** accumulation, which is not mathematically stable under price changes.

3.3 Attack Vector 2 – Abuse of Privileged Roles (Especially FUNDS_RECOVERY_ROLE)

Category: Problems with access control and permission configuration; potential insider threat.

Relevant code paths:

- **recoverBalance(token, user, newBalance)**
- **setBankCapInUsd**, **setWithdrawalLimitInUsd**
- **addToken**, **addTokenPool**
- Role assignments in the constructor

Description

KipuBankV3 uses **AccessControl** to limit operations, but some privileged functions are extremely powerful:

(a) FUNDS_RECOVERY_ROLE – Arbitrary Balance Rewrites

function recoverBalance(address token, address user, uint256 newBalance)

This function:

- Does **not move any tokens**.
- Only changes **internal balances** and **totalBankValueInUsd**.

Malicious scenario (compromised or rogue operator)

1. Attacker obtains the **FUNDS_RECOVERY_ROLE**.
2. Sets **balances[token][attacker]** to a huge value via **recoverBalance**.
3. The contract's on-chain holdings of **token** do not change, but internal **balances** now say the attacker owns all or most of the token.
4. Attacker calls **withdraw(token, amount)** repeatedly until the contract's real token balance is fully drained.

Result: All funds of that token held by the bank can be stolen by the attacker, because internal accounting now grants them claim over all tokens.

This is a direct consequence of extremely powerful admin functions. From a threat-model perspective, it means:

- Loss or compromise of **FUNDS_RECOVERY_ROLE** == total loss of all user funds.

(b) ASSET_MANAGER_ROLE – Misconfigured Oracles and Pools

- **addToken(token, priceFeed)** can point **tokenPriceFeeds[token]** to **any** address.
- **addTokenPool(token, poolKey)** can associate **token** with an arbitrary Uniswap V4 configuration.

If this role is compromised or misused:

- The attacker can point a token to a **malicious or manipulated oracle**, breaking all USD-based controls (caps, limits, etc.).
- The attacker can configure a pool that does not actually map **token -> USDC** as intended, or that routes through malicious hooks.

Impact

- Direct theft (via **FUNDS_RECOVERY_ROLE**).
- Severe mispricing and broken invariants (via **ASSET_MANAGER_ROLE**).
- Trust in the protocol hinges critically on secure, well-managed admin keys and operational processes.

3.4 Attack Vector 3 – Oracle and Token-Behavior Assumption Abuse

Category: Misuse/abuse of protocol assumptions (oracle correctness, “standard” ERC-20 behavior) + economic / griefing strategies.

Relevant code paths:

- **_getPriceUsd8(token)** using Chainlink
- **_getValueInUsd(token, amount)**
- Caps and limits: **bankCapInUsd**, **withdrawalLimitInUsd**
- **deposit** for arbitrary ERC-20 tokens

(a) Oracle Misconfiguration or Manipulation

KipuBankV3 assumes that:

- Each **tokenPriceFeeds[token]** correctly returns the token’s price in USD.
- Chainlink prices are fresh and not easily manipulated.

Griefing scenario via inflated price

1. Attacker convinces an **ASSET_MANAGER_ROLE** to configure a token X with an incorrect price feed (or the oracle itself is compromised).
2. Price feed reports an artificially high price for X, e.g. $10,000.
3. Attacker deposits a relatively small number of X tokens via **deposit(token, amount)**.
4. **_getValueInUsd** returns a huge value; **totalBankValueInUsd** increases sharply and may quickly hit **bankCapInUsd**.

Effects:

- The bank cap is reached with **low actual economic value**, preventing honest users from depositing legitimate assets.
- If prices later "correct" and fall, the logic issues described in Attack Vector 1 can cause further DoS situations.

Even if the attacker cannot directly withdraw more monetary value than they deposited (since withdrawal is in token units), they can **weaponize the bank's risk controls** to block other users from using the protocol.

(b) Non-standard ERC-20 Behavior (Fee-on-Transfer, Rebasing)

The **deposit** function assumes:

```
bool success = IERC20(_tokenAddress).transferFrom(msg.sender, address(this), amount);

…

balances[_tokenAddress][msg.sender] += amount;
```

If the token:

- Charges a **transfer fee**, or
- Burns a portion of tokens on transfer, or
- Is **rebasing** in a way that changes balances after deposit,

then:

- The bank may receive **less** than **amount** tokens, but still credit **balances[token][user] += amount**.
- Later, when the user calls **withdraw**, the contract may not hold enough tokens to honor all internal balances.
- This can lead to insolvency or reverts (depending on how the ERC-20 behaves when transferring more tokens than it actually holds).

Impact

- Insolvency relative to internal accounting.
- Partial or total DoS on withdrawals for that token.
- Silent breaking of solvency invariants (sum of internal balances > on-chain balance).

3.5 Attack Vector 4 – Economic / MEV Interaction with Uniswap & Slippage

Category: Economic/strategic attacks and MEV exploitation.

Relevant code paths:

- **depositArbitraryToken(token, amount, minUsdcOut)**
- **_swapExactInputSingle**

- Uniswap V4 pools and slippage controls

Description

When users call **depositArbitraryToken**, KipuBankV3:

1. Transfers **token** from the user (or wraps ETH to WETH).
2. Swaps **token ->USDC** via Uniswap V4 using the Universal Router.
3. Credits the user with **actual** USDC received, as long as **amountOut >= minUsdcOut**.

From a correctness standpoint, this is reasonable. However, it introduces typical **DeFi / MEV** risks:

(a) Front-running Around Swaps

- A user submits **depositArbitraryToken** with a tight **minUsdcOut**.
- A MEV bot sees this transaction in the mempool and:
    - Moves the price in the relevant pool (e.g., via a large swap) just before the user's tx.
    - Causes the actual **amountOut** to be below **minUsdcOut**.
- The user's transaction reverts; the user loses gas and may struggle to get a successful deposit.

This is not a direct theft of funds (since the whole tx reverts), but it **degrades UX** and can be used as griefing.

(b) Misconfigured or Malicious Router / Pool

If:

- **universalRouter** is set to an address that is not the genuine Uniswap V4 router, or
- **tokenToUsdcPools[token]** points to a dangerous or manipulated pool,

then:

- The router might execute arbitrary behavior with the bank's tokens (e.g., routing through hooks that siphon value).
- The bank may receive far less USDC than expected, especially if **minUsdcOut** is set too low by the user.

While the contract does **not** credit more than it actually receives, a malicious/swapped router could drain tokens from the bank during repeated deposits, using the bank itself as the "payer" in the router command (**payerIsUser = false**).

Impact

- Value extraction from the bank's reserves if the router is not the genuine Uniswap implementation.

- Increased susceptibility to MEV griefing for users with tight slippage bounds.

3.6 Summary

In summary, KipuBankV3's main risks cluster around business-logic consistency (**totalBankValueInUsd**), oracle and token-behavior assumptions, MEV and DEX-related economics, and the power of privileged roles. These themes directly motivate the invariants and recommendations presented in the next sections.

4. Invariant Specification

This section defines protocol-level invariants that KipuBankV3 **should** preserve in every valid state transition. They are the basis for both security reasoning and property-based testing.

4.1 Invariant #1 – Per-Token Solvency

For every supported token **T** (including ETH via **ETH_ADDRESS**), the contract must never promise users more than it actually holds.

Let:

- **ContractBalance(T)** = on-chain balance of token **T** held by **KipuBankV3** (**IERC20(T).balanceOf(address(this))** or **address(this).balance** for ETH),
- **InternalBalance(T)** = Sum **balances[T][u]** over all users **u**.

Invariant:

For all tokens **T**: **ContractBalance(T) ≥ InternalBalance(T).**

This is the fundamental "bank is solvent per asset" property.

4.2 Invariant #2 – Aggregate USD Value Consistent and Within Cap

The variable **totalBankValueInUsd** is intended to approximate the bank's total USD exposure. Ignoring oracle failures, the desired properties are:

- **Non-negativity:** **totalBankValueInUsd ≥ 0** at all times.
- **Respecting the configured :** After any successful state-changing operation: **totalBankValueInUsd ≤ bankCapInUsd**.

No deposit, swap-based deposit, or balance recovery should leave the bank with a reported value above the configured cap, and the value must not underflow due to arithmetic or pricing issues.

The current implementation updates **totalBankValueInUsd** using "current" prices in both deposits and withdrawals, which is precisely what makes this invariant fragile (see Attack Vector 1).

4.3 Invariant #3 – Withdrawal Correctness and Limit Enforcement

For any successful call to**: withdraw(token, amount)**

two things must always hold:

- **Accounting correctness**
  Let **u** = **msg.sende**r. Then:
  - Pre-condition: **balances[token][u] ≥ amou**nt,
  - Post-condition: **balances[token][u]_new = balances[token][u]_old − amount**,
  - The contract transfers exactly **amount** of **token** (or ETH) to **u**.

- **Per-transaction USD limit**
  Let **amountInUsd = _getValueInUsd(token, amount)** (using the current price feed).
  Then: **amountInUsd ≤ withdrawalLimitInUsd** for all successful withdrawals.

No user should be able to withdraw more than their internal balance, nor more than the configured USD limit in a single transaction.

5. Impact of Invariant Violations

This section links the invariants above to their security impact, referencing the attack vectors described in Section 3.

5.1 Invariant #1 – Token Solvency

If for some token **T** we have **ContractBalance(T) < InternalBalance(T)**:

- The protocol becomes partially or fully **insolvent** for T: it cannot honor all user balances.
- Early withdrawers may succeed while late users are unable to withdraw, effectively losing funds.
- This can arise from:
  - Misuse of **recoverBalance** (e.g., inflating a user's balance without corresponding deposits),
    Interacting with non-standard ERC-20s (fee-on-transfer, rebasing) without adjusting accounting.

This is the **most severe** failure mode: it directly maps to user fund loss and "bank run" dynamics.

5.2 Invariant #2 – Aggregate USD Value

Breaking this invariant has two main manifestations, closely related to Attack Vector 1:

> **Underflow / negative value (DoS on withdrawals)**
> If **totalBankValueInUsd** is updated using current prices and a token's price rises between deposit and withdrawal, the subtraction in **withdraw** can attempt to subtract more than the stored value. This causes reverts, leading to:
>
> - Denial of Service on withdrawals for that token,
> - Potential griefing if an adversary can manipulate the oracle to trigger this behavior.
>
> **Exceeding bankCapInUsd silently**
> If **recoverBalance** is used to increase balances (and thus **totalBankValueInUsd**) without re-enforcing the cap:
>
> - The reported exposure no longer respects the configured risk bound,
> - Any off-chain monitoring that relies on **totalBankValueInUsd** becomes misleading.

In both cases, the variable loses its meaning as a reliable risk indicator, and can even block normal operation.

5.3 Invariant #3 – Withdrawals and Limits

If withdraw logic fails to maintain this invariant:

- **Incorrect accounting**
  If balances are not reduced correctly, users may withdraw multiple times against the same recorded balance (a free-money bug), draining liquidity.
- **Bypassing the USD limit**
  If **withdrawalLimitInUsd** is not correctly enforced, a single transaction could withdraw an excessively large amount of value, contrary to the risk model.
- **Inconsistent state**
  Any mismatch between debited balances and transferred tokens complicates reconciliation and can cascade into solvency issues (violating Invariant #1).

The net effect is either direct fund loss or uncontrolled liquidity shocks.

6. Recommendations – Validating and Enforcing Invariants

This section focuses specifically on **how to validate and enforce** the invariants in practice, both via tests and operational measures.

6.1 Validating Solvency (Invariant #1)

Code and design

- Restrict the set of supported tokens to those with standard ERC-20 behavior (no fees, no rebasing), or handle them explicitly.
- Constrain **recoverBalance** so that:
    - Increasing a user's balance is only allowed if **ContractBalance(T) ≥ newBalance** for that user (or for the total across users), and/or
    - Upward corrections require stronger process (e.g., multi-sig + timelock).

Testing

- Add Foundry tests that:
    - Perform sequences of deposits and withdrawals across multiple users and verify:
    - Exercise **recoverBalance** in both directions and check solvency after each operation.
- Add property-based / invariant tests (e.g., Foundry **invariant_** tests or Echidna) where the fuzzer calls deposit/withdraw/recoverBalance and the invariant:

  For all T: **ContractBalance(T) ≥ InternalBalance(T)** must hold after each sequence.

- Optionally, build an off-chain monitor that periodically recomputes solvency using on-chain data.

6.2 Validating Aggregate USD Bounds (Invariant #2)

Design improvements

- Prefer computing total USD value **on demand** from per-token balances and oracles in a view function, instead of maintaining a mutable **totalBankValueInUsd**.
  This removes a whole class of underflow and drift issues.
- If a mutable aggregate is kept:
    - Add explicit checks after each update:
        - **require(totalBankValueInUsd <= bankCapInUsd, "Bank cap violated");**
    - Ensure **recoverBalance** cannot increase **totalBankValueInUsd** beyond **bankCapInUsd**.

Testing

- Add tests that simulate price movements between deposit and withdrawal (by changing mock oracle answers) and assert:
    - Withdrawals do not revert solely due to price moves when the system is otherwise solvent,
    - **totalBankValueInUsd** does not underflow or exceed **bankCapInUsd** after successful operations.
- Add an invariant test:
  0 ≤ totalBankValueInUsd ≤ bankCapInUsd under realistic oracle ranges.

6.3 Validating Withdrawals and Limits (Invariant #3)

Unit tests

- Add explicit tests for:
    - Withdraw reverting with **InsufficientBalance**,
    - Withdraw reverting with **WithdrawalAmountExceedsUsdLimit**,
    - Boundary conditions where **amountInUsd == withdrawalLimitInUsd**.

Property-based tests

- Use fuzzing to generate random sequences of deposits and withdrawals for multiple users, and assert that:
    - Balances never go negative,
    - No withdrawal succeeds with **amountInUsd > withdrawalLimitInUsd**.