Threat and Invariant Analysis Report: KipuBankV3

Date: December 12, 2025
Subject: KipuBankV3
Protocol Network: Ethereum Mainnet
Author: Lorenzo Pereira Piccoli Xavier

1. Protocol Overview

KipuBankV3 is a **decentralized savings** and vault protocol designed to accept deposits in Native ETH, ERC20 tokens, and arbitrary tokens through a swap mechanism. Compiled with Solidity version 0.8.26, the protocol integrates with **Uniswap V4** for liquidity and **Chainlink Oracles** for on-chain asset valuation, in addition to relying on **OpenZeppelin** for role definition.

Key Mechanisms:

- **Deposits:** Users can deposit Native ETH or whitelisted ERC20 tokens directly.
- **Swaps:** The `depositArbitraryToken` function utilizes the Uniswap V4 PoolManager and Permit2 to swap user tokens for a base currency (USDC) prior to deposit, utilizing "Flash Accounting" via `unlockCallback`.
- **Risk Management:** The protocol imposes a Global TVL (Total Value Locked) Cap denominated in USD ($1,000,000) and individual withdrawal limits for native ETH.
- **Accounting:** Balances are tracked per user and per token. The protocol dynamically calculates the bank's total value using oracle data feeds provided by Chainlink.

With the use of Solidity Metrics, we are able to quantitatively visualize some aspects of the contract.

File	Logic Contracts	Lines	nLines	nSLOC	Comment Lines	Complex. Score
src\Kipu.sol	1	507	486	288	129	247

In our 507-line contract, 129 lines are dedicated to comments, which accounts for almost 20% of the total lines, indicating an effort to ensure the contract is clear and well-documented. Metrics used to measure code complexity, excluding comments and blank

lines, include an nSLOC (Normalized Source Lines of Code) of 288 and a Complexity Score of 247. This indicates that half of the written lines of code perform executable instructions.

1.1 Imported Dependencies

It is crucial to identify which dependencies have been imported, as a vulnerability could arise that renders the contract insecure. Solidity Metrics generated the following **Table 1**, listing all imports included in KipuBankV3.

Dependências	Contagem
@chainlink/contracts/src/v0.8/shared/interfaces/AggregatorV3Interface.sol	1
@openzeppelin/contracts/access/AccessControl.sol	1
@openzeppelin/contracts/token/ERC20/ERC20.sol	1
@openzeppelin/contracts/token/ERC20/IERC20.sol	1
@openzeppelin/contracts/token/ERC20/extensions/ERC20Pausable.sol	1
@openzeppelin/contracts/token/ERC20/extensions/ERC20Permit.sol	1
@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol	1
@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol	1
@openzeppelin/contracts/utils/Pausable.sol	1
@uniswap/universal-router/contracts/interfaces/IUniversalRouter.sol	1
@uniswap/v4-core/src/interfaces/IHooks.sol	1
@uniswap/v4-core/src/interfaces/IPoolManager.sol	1
@uniswap/v4-core/src/interfaces/callback/IUnlockCallback.sol	1
@uniswap/v4-core/src/libraries/Hooks.sol	1
@uniswap/v4-core/src/types/BalanceDelta.sol	1
@uniswap/v4-core/src/types/Currency.sol	1
@uniswap/v4-core/src/types/PoolKey.sol	1
@uniswap/v4-core/src/types/PoolOperation.sol	1
@uniswap/v4-periphery/lib/permit2/src/interfaces/IPermit2.sol	1

@uniswap/v4-periphery/src/base/DeltaResolver.sol	1
@uniswap/v4-periphery/src/base/SafeCallback.sol	1
@uniswap/v4-periphery/src/utils/BaseHook.sol	1

1.2 Funções do contrato

Using the Solidity Metrics tool, a breakdown of all contract functions was generated. Note: Getter methods for public state variables were not included.

Public	Payable
22	5

It is observed that the majority of the contract's functions are public, while a small portion are `payable` (`receive()`, `depositNative()`, `depositERC20()`, `SetannotationBank()`, and `fallback()`). These `payable` functions require caution during the audit, necessitating unit and invariant tests.

External	Internal	Private	Pure	View
13	14	4	1	9

It is observed that the majority of functions are internal and external; a significant number of `view` functions is also noted, which makes sense from a banking perspective.

2. Protocol Maturity Assessment

Before proceeding to an external audit, the protocol currently presents a **Low-to-Medium** maturity level. Below is the gap analysis:

A. Coverage and Test Methods

- **Current State:** Tests rely heavily on manual scripts (`Kipu.t.sol`) and basic interaction scripts.
- **Gaps:** Lack of comprehensive unit tests covering edge cases. Specifically, Fuzz Testing (Stateless) and, most importantly, Invariant Testing (Stateful) are absent.

- **Recommendation:** Implement a Foundry test suite achieving >95% branch coverage. Use `forge test --fuzz-runs` to verify arithmetic overflows and `forge test invariant` to validate the Bank Cap logic over thousands of random transaction sequences.

B. Current Documentation

- **Current State:** The code contains in-line comments but lacks high-level architectural documentation. The lack of helpers is a concern, as it may compromise the understanding of the smart contract by third parties.
- **Gaps:** Absence of NatSpec (Natural Specification) for public functions. Lack of sequence diagrams explaining the interaction between KipuSafe, Permit2, and the PoolManager.
- **Recommendation:** Generate automatic documentation using `forge doc` or the Solidity Metrics extension (VS Code). Create a diagram illustrating the `unlockCallback` flow to ensure auditors understand the trust assumptions with the PoolManager.

C. Roles and Powers (Access Control)

- **DEFAULT_ADMIN_ROLE:** Can grant/revoke roles. **Risk: Critical.** If compromised, the entire access control system fails.
- **PAUSER_ROLE:** Can freeze all deposits and withdrawals (`pause()`, `unpause()`). **Risk: High (Denial of Service).**
- **ownerContract:** Can change the owner's address. **Risk: Medium.**

Observation: The code currently deploys these roles to an EOA (Externally Owned Account/regular wallet) in the constructor. For Mainnet, these must be transferred to a **Multi-sig Wallet (Safe)**, as done in the first module of the course, or a **Timelock Controller**, which works to prevent malicious actions from being executed immediately in the event of wallet hijacking.

3. Attack Vectors and Threat Model

Identificadas três superfícies de ataque críticas com base na lógica atual:

Scenario A: Oracle Manipulation and Stale Data (Economic Exploitation)

- **Vulnerability:** The functions `getETH2USD()` and `getTOKEN2USD()` rely on `dataFeed.latestRoundData()` provided by Chainlink. Although the code checks for negative prices, it does not currently check if the data is stale (stale data check on `updatedAt`).
- **Attack Vector:** If the Chainlink oracle stops updating (or the L2 sequencer goes down) during a market crash, the contract will "think" that ETH is worth more than it actually is. A malicious user could deposit ETH at the old (high) price, filling the BankCap with devalued assets, effectively diluting the real value of the protocol.

- **Mitigation:** Implement checks for `block.timestamp - updatedAt <= heartbeat`.

Scenario B: Sandwich Attacks on Swaps (MEV)

- **Vulnerability:** In the `depositArbitraryToken` function, the user passes `amountOutMin`. If the frontend or the user sets this to 0 (or a very low number) to avoid transaction failures, the system becomes vulnerable.
- **Attack Vector:** An MEV bot detects the pending transaction in the mempool. They front-run (buy before) to push the price of `tokenIn` down against USDC, and then back-run. The protocol receives significantly less USDC than the market rate, but the user is credited for the full deposit "effort" (depending on how internal balances are updated vs actual values received).
- **Mitigation:** The UI must enforce a rigorous slippage calculation. The contract logic correctly checks `amountOut >= amountOutMin`, but security relies entirely on the input parameter being correct.

Scenario C: Bank Cap Griefing (Denial of Service)

- **Vulnerability:** `BANK_MAX_CAP_USDC` is a shared global state variable.
- **Attack Vector:** A malicious competitor or hostile actor could use a flash-loan of a large amount of assets and deposit them into KipuSafe to hit exactly the $1,000,000 cap.
- **Impact:** Legitimate users are unable to deposit funds (revert with `MaxBankCapReached`). The attacker can keep the funds there (earning nothing, but blocking others) or withdraw and repeat the process.
- **Mitigation:** Implement a "Whitelisted Depositor" mode for the initial phase or a "Deposit Cap per User" to prevent a single whale from filling the entire cap.

4. Invariant Specification

As seguintes propriedades devem permanecer verdadeiras para o sistema **em todos os momentos**, independentemente das ações do usuário ou volatilidade do mercado.

Invariant 1: The Solvency Invariant

"The contract's actual token balance (held at the address) must always be greater than or equal to the sum of all individual user balances recorded in the balances mapping."

- **Logic:** The bank cannot owe users more money than it physically possesses.

Invariant 2: The Global Cap Invariant

"The Total Value Locked (TVL) of the protocol, converted to USD using the current Oracle price, must never exceed `BANK_MAX_CAP_USDC` after a deposit action."

- ***Logic:*** *This ensures that risk exposure is limited to the defined threshold ($1M).* ***Note:*** *Market fluctuations may raise the value post-deposit, but no new deposit should be accepted if Current TVL > Cap.*

Invariant 3: The WETH/Native Isolation

> *"Deposits made via depositNative must increment balances[address(0)], and withdrawals via withdrawNative must decrement it. Native ETH balances must never be mixed with Wrapped ETH (WETH) balances in internal accounting."*

- **Logic:** Prevents accounting errors where users deposit ETH but attempt to withdraw WETH without a wrapping mechanism, or vice versa.

5. Impact of Invariant Violations

- **Solvency Violation (Inv 1):** Catastrophic failure. If the contract holds less than the recorded liabilities, it is essentially a fractional reserve bank with no reserves. The last users to withdraw will lose their funds (Bank Run).
- **Global Cap Violation (Inv 2):** Breakdown of risk management. If the cap is bypassed, the protocol may manage more capital than its security maturity allows, becoming an attractive "honeypot" for hackers.
- **Isolation Violation (Inv 3):** Locked funds. Users may see a balance in the UI, but the contract will revert when trying to transfer the wrong asset type, leading to stuck funds and loss of trust.

6. Recommendations

To validate these invariants and prepare for the professional audit:

1. **Implement Stateful Fuzzing (Foundry):** Create a `Handler.sol` contract that performs random deposits, withdrawals, and swaps. Perform the test below aiming to ensure the bank never reaches a state where it lacks sufficient liquidity in the event that all users wish to withdraw their funds deposited in the smart contract. Subsequently, write an `Invariant.sol` contract that asserts: `function invariant_solvency() public {assertGe(IERC20(USDC).balanceOf(address(kipu)), totalUserBalances);}`
2. **Oracle Mock:** To test Invariant 2 (Cap), create a Mock Oracle (fake) that drastically alters the ETH price. Ensure that, even if ETH goes to $10,000, new deposits revert if the new calculated cap exceeds the limit.
3. **Formal Verification:** Consider using tools like Halmos (symbolic testing) to mathematically prove that `balances[msg.sender]` can never decrease without a `withdraw` call.

After using Halmos, we obtained the following result:

Categoria	Count
unused-import	11
unaliased-import	6
mixed-case-function	12
mixed-case-variable	6
screaming-snake-case (const + immutable)	8
erc20-unchecked-transfer	2
anothers	2

Indicating that certain measures must be taken for the contract to be ready for a professional audit.

4. Use of Slither (https://github.com/crytic/slither) to verify possible security flaws; according to documentation, it features over 99 detectors ranging from severe security risks to optimization..

After using Slither, we obtained the following result: Some false positives in already audited code such as OpenZeppelin and Uniswap. However, after their removal, we observed the following:

Severidade	Quantidade	Ação
Critic	3	Solve immediately
Medium	3	Adjust or documment
False Positive	~40	Ignore

INFO:Slither:src/ analyzed (88 contracts with 100 detectors), 228 result(s) found

5. Utilization of Tenderly for validation of the entire contract logic on a real fork of the network where deployment is intended.
6. After creating all necessary tests—whether unit or invariant, happy path or otherwise—perform a forge test.

7. Conclusion and Next Steps

The **KipuBankV3** protocol demonstrates a sophisticated integration with **Uniswap V4** and **Permit2**, but currently relies on "*Happy Path*" assumptions. The lack of automated invariant tests and the potential for stale Oracle data are the biggest blockers for a Mainnet launch.

Immediate Next Steps:

1. **Refactor Oracle Logic:** Add checks for `updatedAt` and stale prices.
2. **Write Invariant Tests:** Implement Solvency and Cap checks in Foundry.
3. **Deploy on Testnet (Sepolia) with Liquidity:** Ensure a real Uniswap V4 Pool exists (add liquidity) to validate the `depositArbitraryToken` flow end-to-end.
4. **Transfer Ownership:** Move administrative roles to a Multi-Sig (Gnosis Safe).

Once these steps are completed, KipuBankV3 will be ready for a professional security audit.