AstraSec

Layerbank V3 Security Audit Report

July 17, 2025

Contents

1 Introduction

1.1 About Layerbank V3

LayerBank V3 is a decentralized lending protocol built on a robust and battle-tested architecture. Following a strategic adjustment, the LayerBank team decided to migrate its EVM-based infrastructure from the previous implementation to a new design focused on enhancing security, modularity, and scalability. The protocol features a complete suite of lending functionalities, including lending pools, interest-bearing tokens, dynamic interest rate models, secure liquidation mechanisms, and strategy contracts that facilitate users to create leveraged positions. LayerBank V3 is positioned as the core infrastructure layer to support future protocol deployments and innovations within the LayerBank ecosystem.

1.2 Source Code

The following source code was reviewed during the audit:

- https://github.com/layerbank-foundation/v3-contracts-audit
- Commit: 32e2517 & 7849564

It should be noted that the audit scope under extensions directory only covers the the following directories:

src/contracts/extensions/leverage-looping
src/contracts/extensions/maverick-adapter
src/contracts/extensions/morpho-adapter
src/contracts/extensions/nest-adapter
src/contracts/extensions/leverage-looping/strategies/internal

This is the final version representing all fixes implemented for the issues identified in the audit:

- layerbank-foundation/v3-contracts-audit@fix/leverage-looping-findings
- Commit: aeefeed

1.3 Revision History

Version	Date	Description
v1.0	July 17, 2025	Initial Audit

2 Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the Layerbank V3 protocol. Throughout this audit, we identified a total of 7 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	–	–	–	–
High	1	–	–	1
Medium	4	1	–	3
Low	1	–	–	1
Informational	1	1	–	–
Undetermined	–	–	–	–

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

~~H-1~~	Incorrect Implementation Logic in getBestPool()
~~M-1~~	Invalid Slippage Control in MaverickSwapAdapter
~~M-2~~	Hardcoded Slippage in MaverickLeverageStrategy
~~M-3~~	Flashloan Amount Miscalculation in Aave Related Strategies
M-4	Potential Risks Associated with Centralization
~~L-1~~	Revisited Implementation Logic in MaverickLeverageStrategy
I-1	Suggested Revert Usages For Gas Efficiency

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

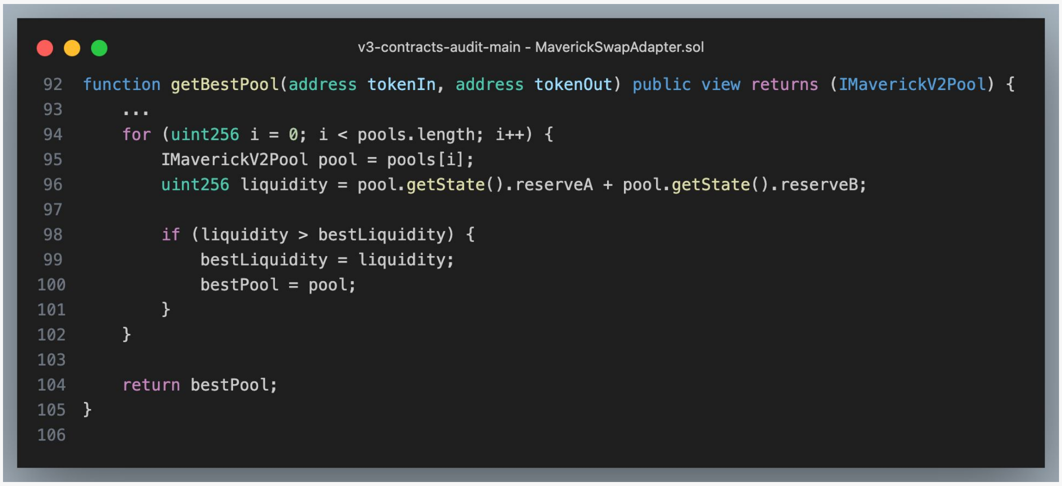
Severity	Acknowledged
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and remediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3 Vulnerability Details

3.3.1 [H-1] Incorrect Implementation Logic in getBestPool()

Target	Category	IMPACT	LIKELIHOOD	STATUS
MaverickAdapter	Business Logic	High	Low	Addressed

The getBestPool() function in the MaverickSwapAdapter contract is used to search all pools for a given token pair (tokenIn and tokenOut) from the Maverick V2 factory contract and return the one with the highest liquidity. It does so by calling the factory's lookup function to retrieve a list of pools, iterating through each pool, calculating the total reserves (reserveA + reserveB), and selecting the pool with the highest total reserves. However, the current implementation does not account for the possibility that tokenIn and tokenOut may have different decimals. Moreover, this approach to select the best pool by comparing reserveA + reserveB is not suitable for concentrated liquidity AMMs like Maverick V2. It fails to account for the actual liquidity available near the current price range and does not consider trade direction or price impact, which may result in suboptimal pool selection and poor execution prices.


v3-contracts-audit-main - MaverickSwapAdapter.sol

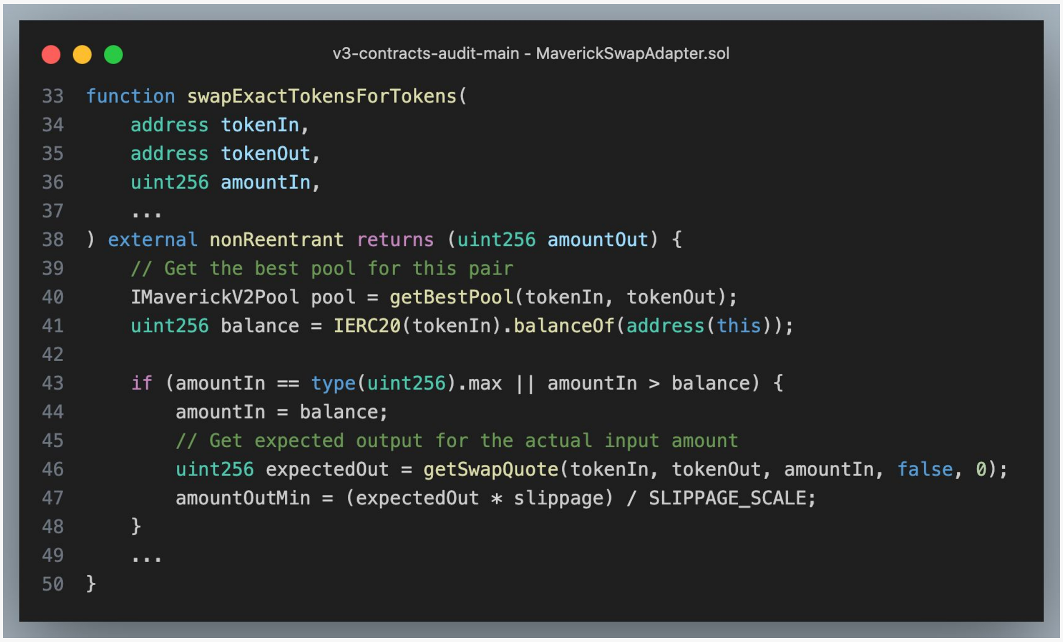
```
function getBestPool(address tokenIn, address tokenOut) public view returns (IMaverickV2Pool) {
    ...
    for (uint256 i = 0; i < pools.length; i++) {
        IMaverickV2Pool pool = pools[i];
        uint256 liquidity = pool.getState().reserveA + pool.getState().reserveB;

        if (liquidity > bestLiquidity) {
            bestLiquidity = liquidity;
            bestPool = pool;
        }
    }

    return bestPool;
}
```

Remediation It is recommended to simulate swap quotes for each candidate pool and select the one offering the best effective rate.

3.3.2 [M-1] Invalid Slippage Control in MaverickSwapAdapter

Target	Category	IMPACT	LIKELIHOOD	STATUS
MaverickAdapter	Business Logic	High	Low	Addressed

The helper function swapExactTokensForTokens() in the MaverickSwapAdapter contract is designed to swap a specified amount of tokenIn for tokenOut on Maverick V2, while ensuring that the amount of tokenOut received is not less than a predefined minimum. During our review of the function's implementation, we notice that when the user passes amountIn as the maximum uint256 value (i.e., type(uint256).max) or provides an amount exceeding the contract's balance, the function automatically adjusts the input amount to the contract's actual balance. It then calls getSwapQuote() to retrieve the current quote and dynamically calculates amountOutMin based on that quote. However, since the quoting function relies on the on-chain state, it is vulnerable to manipulation by MEV bots, which can result in an execution price significantly worse than expected, thereby rendering the slippage protection ineffective.



v3-contracts-audit-main - MaverickSwapAdapter.sol

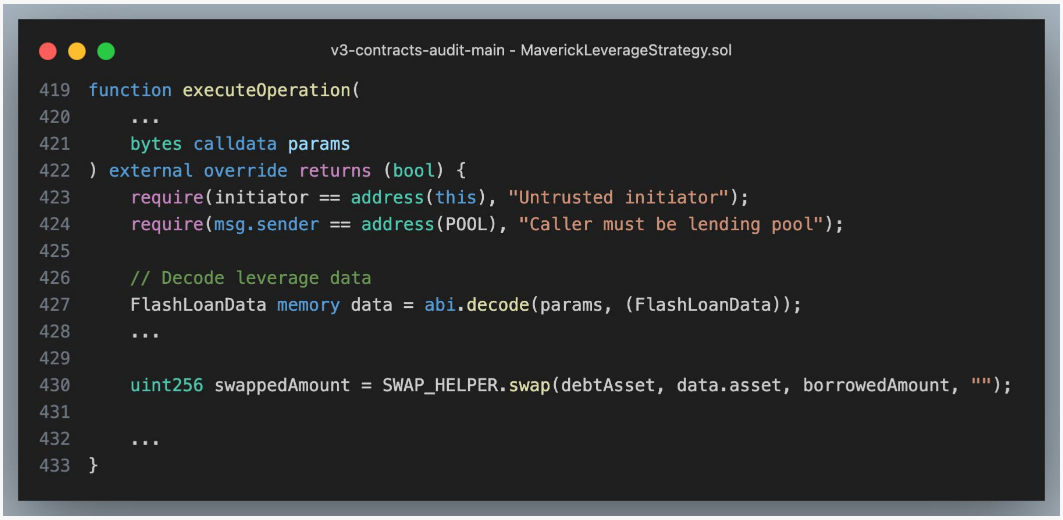
```
33 function swapExactTokensForTokens(
34     address tokenIn,
35     address tokenOut,
36     uint256 amountIn,
37     ...
38 ) external nonReentrant returns (uint256 amountOut) {
39     // Get the best pool for this pair
40     IMaverickV2Pool pool = getBestPool(tokenIn, tokenOut);
41     uint256 balance = IERC20(tokenIn).balanceOf(address(this));
42 
43     if (amountIn == type(uint256).max || amountIn > balance) {
44         amountIn = balance;
45         // Get expected output for the actual input amount
46         uint256 expectedOut = getSwapQuote(tokenIn, tokenOut, amountIn, false, 0);
47         amountOutMin = (expectedOut * slippage) / SLIPPAGE_SCALE;
48     }
49     ...
50 }
```

Remediation To mitigate potential sandwich attacks in the input scenarios described above, it is recommended to remove the code from lines 46 to 51.

3.3.3 [M-2] Hardcoded Slippage in MaverickLeverageStrategy

Target	Category	IMPACT	LIKELIHOOD	STATUS
MaverickLeverage	Business Logic	Medium	Medium	Addressed

In the executeOperation() function, when performing the token swap via the SWAP_HELPER contract, the slippage control parameter data.slippage—decoded from the FlashLoanData struct—is not utilized. Instead, the swap is executed using a hardcoded 3% slippage threshold internally within the SWAP_HELPER. This implementation introduces several potential risks. First, the hardcoded 3% slippage threshold lacks flexibility and cannot be dynamically adjusted based on market volatility or trade size, which may lead to unnecessary transaction costs or failures in highly volatile markets. Second, the fixed slippage setting may create arbitrage opportunities for malicious actors, such as profiting within the 3% range by manipulating prices or inserting transactions (front-running), thereby increasing the cost of execution for regular users.


v3-contracts-audit-main - MaverickLeverageStrategy.sol

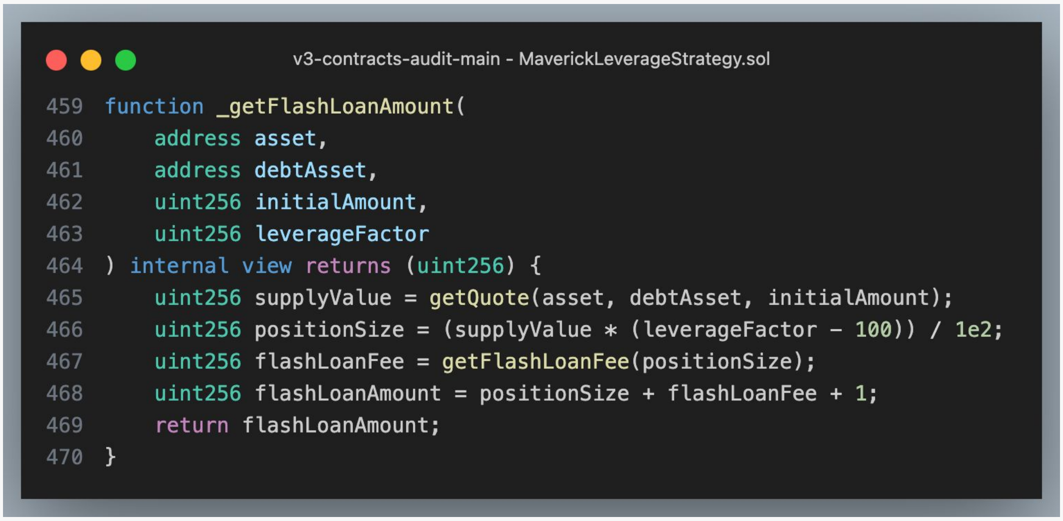
```
function executeOperation(
    ...
    bytes calldata params
) external override returns (bool) {
    require(initiator == address(this), "Untrusted initiator");
    require(msg.sender == address(POOL), "Caller must be lending pool");

    // Decode leverage data
    FlashLoanData memory data = abi.decode(params, (FlashLoanData));
    ...

    uint256 swappedAmount = SWAP_HELPER.swap(debtAsset, data.asset, borrowedAmount, "");

    ...
}
```

Remediation Refactor the swap() function call to explicitly pass and enforce the data.slippage parameter during the swap operation. This ensures users retain control over acceptable price deviations and enhances the protocol’s resistance to front-running and adverse price movements.

3.3.4 [M-3] Flashloan Amount Miscalculation in Aave Related Strategies

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Business Logic	Low	High	Addressed

In Aave V3, InterestRateMode indicates the interest model applied when borrowing assets. When InterestRateMode != NONE, the borrower opts not to repay the flashloan within the same transaction but instead opens a debt position under the chosen rate model. In the leverage strategy contracts reviewed, all AAVE related flashloan operations specify VARIABLE as the InterestRateMode, meaning the borrowed amount is converted into variable-rate debt and no flashloan fee is charged. However, these strategy contracts incorrectly factor in the flashloan premium when calculating the flashloan amount, despite no fee being applicable in this mode. This miscalculation can lead to under-leveraged positions and inefficient capital usage, deviating from intended leverage ratios.


v3-contracts-audit-main - MaverickLeverageStrategy.sol

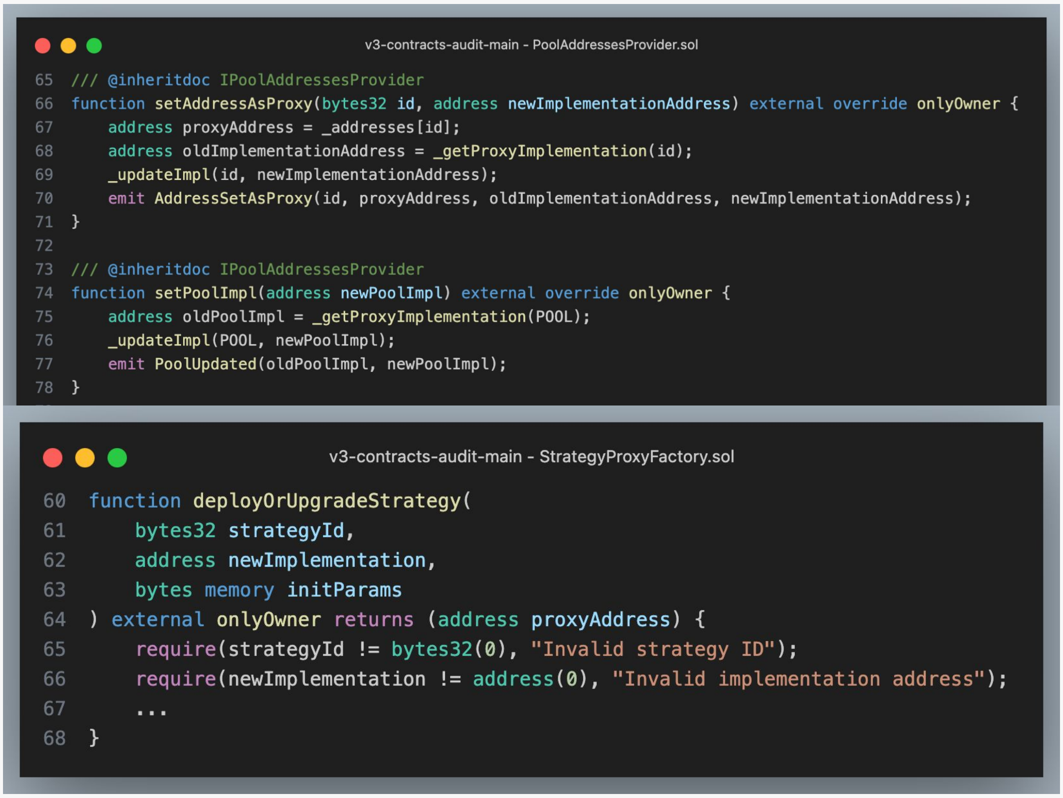
```solidity
function _getFlashLoanAmount(
    address asset,
    address debtAsset,
    uint256 initialAmount,
    uint256 leverageFactor
) internal view returns (uint256) {
    uint256 supplyValue = getQuote(asset, debtAsset, initialAmount);
    uint256 positionSize = (supplyValue * (leverageFactor - 100)) / 1e2;
    uint256 flashLoanFee = getFlashLoanFee(positionSize);
    uint256 flashLoanAmount = positionSize + flashLoanFee + 1;
    return flashLoanAmount;
}
```

Remediation Update the flashloan amount calculation logic to exclude the flashloan premium when InterestRateMode is set to VARIABLE, as no fee applies in this case. This ensures accurate leverage behavior and protocol compliance.

3.3.5 [M-4] Potential Risks Associated with Centralization

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Security	High	Low	Acknowledged

In the Layerbank V3 project, the existence of a privileged owner account introduces centralization risks, as it holds significant control and authority over critical operations governing the protocol. In the following, we show the representative function potentially affected by the privileges associated with the privileged account.



v3-contracts-audit-main - PoolAddressesProvider.sol

```
/// @inheritdoc IPoolAddressesProvider
function setAddressAsProxy(bytes32 id, address newImplementationAddress) external override onlyOwner {
    address proxyAddress = _addresses[id];
    address oldImplementationAddress = _getProxyImplementation(id);
    _updateImpl(id, newImplementationAddress);
    emit AddressSetAsProxy(id, proxyAddress, oldImplementationAddress, newImplementationAddress);
}

/// @inheritdoc IPoolAddressesProvider
function setPoolImpl(address newPoolImpl) external override onlyOwner {
    address oldPoolImpl = _getProxyImplementation(POOL);
    _updateImpl(POOL, newPoolImpl);
    emit PoolUpdated(oldPoolImpl, newPoolImpl);
}
```

v3-contracts-audit-main - StrategyProxyFactory.sol

```
function deployOrUpgradeStrategy(
    bytes32 strategyId,
    address newImplementation,
    bytes memory initParams
) external onlyOwner returns (address proxyAddress) {
    require(strategyId != bytes32(0), "Invalid strategy ID");
    require(newImplementation != address(0), "Invalid implementation address");
    ...
}
```

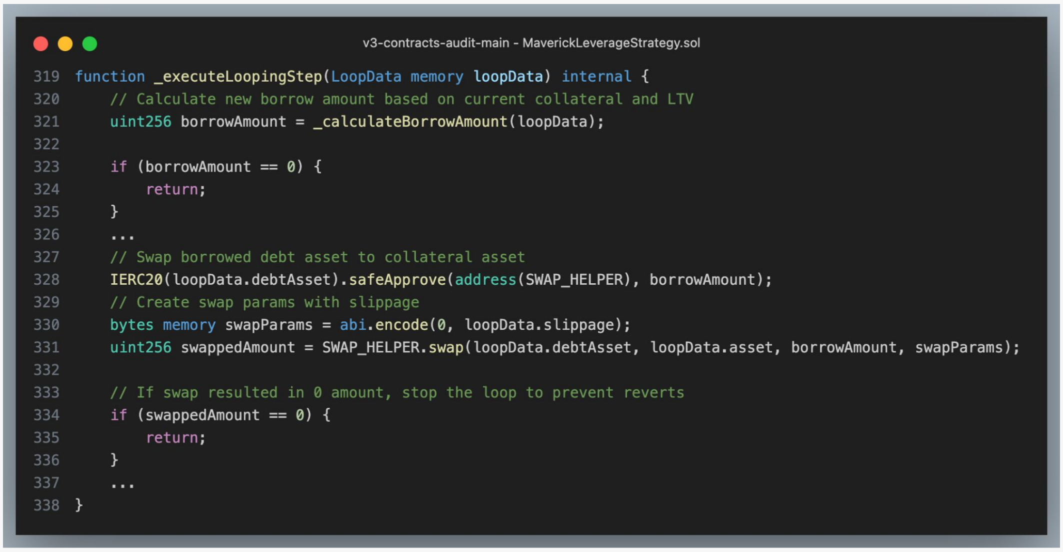
Remediation To mitigate the identified issue, it is recommended to introduce multi-sig mechanism to undertake the role of the privileged account. Moreover, it is advisable to implement timelocks to govern all modifications to the privileged operations.

Response By Team This issue has been confirmed by the team.

3.3.6 [L-1] Revisited Implementation Logic in MaverickLeverageStrategy

Target	Category	IMPACT	LIKELIHOOD	STATUS
MaverickLeverage	Business Logic	Medium	Low	Addressed

In the _executeLoopingStep() function, when the token swap via SWAP_HELPER returns a swappedAmount of zero, the function immediately exits without updating the corresponding loopData.totalBorrowed value. This behavior introduces a logical inconsistency in the leverage strategy execution: the debt has already been borrowed from the Aave pool, but since totalBorrowed is not updated, the leverage tracking becomes inaccurate. This can affect subsequent loop iterations and mislead downstream logic, potentially leading to miscalculated leverage positions.



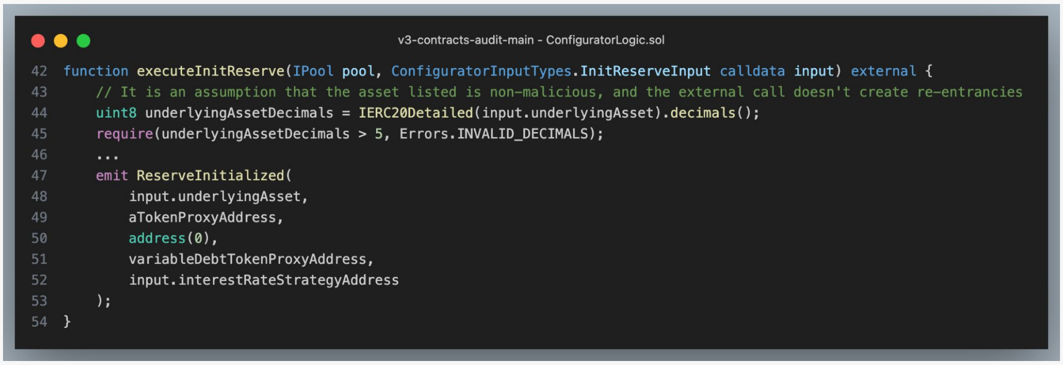
```solidity
function _executeLoopingStep(LoopData memory loopData) internal {
    // Calculate new borrow amount based on current collateral and LTV
    uint256 borrowAmount = _calculateBorrowAmount(loopData);

    if (borrowAmount == 0) {
        return;
    }
    ...
    // Swap borrowed debt asset to collateral asset
    IERC20(loopData.debtAsset).safeApprove(address(SWAP_HELPER), borrowAmount);
    // Create swap params with slippage
    bytes memory swapParams = abi.encode(0, loopData.slippage);
    uint256 swappedAmount = SWAP_HELPER.swap(loopData.debtAsset, loopData.asset, borrowAmount, swapParams);

    // If swap resulted in 0 amount, stop the loop to prevent reverts
    if (swappedAmount == 0) {
        return;
    }
    ...
}
```

Remediation Ensure that loopData.totalBorrowed is updated even when the swap fails (i.e., swappedAmount == 0), or implement a rollback mechanism to reverse the borrow action. This will maintain internal state consistency and enhance the robustness of the leverage strategy.

3.3.7 [I-1] Suggested Revert Usages For Gas Efficiency

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Coding Practices	N/A	N/A	Acknowledged

The LayerBank V3 protocol codebase extensively uses require statements for input validation and error handling, rather than utilizing revert with custom error types or more gas-efficient alternatives. While require offers straightforward conditional checks, its frequent usage throughout the code can lead to higher gas consumption, particularly in complex functions with multiple validation steps. In contrast, modern versions of Solidity support revert with custom errors, which encode error data more efficiently and avoid unnecessary string storage, resulting in significantly lower gas costs. Replacing some require statements with revert and custom errors can enhance gas efficiency and overall contract performance, especially under high-frequency usage scenarios. This optimization aligns with best practices in smart contract development and helps reduce transaction costs for end users.


v3-contracts-audit-main - ConfiguratorLogic.sol

```solidity
function executeInitReserve(IPool pool, ConfiguratorInputTypes.InitReserveInput calldata input) external {
    // It is an assumption that the asset listed is non-malicious, and the external call doesn't create re-entrancies
    uint8 underlyingAssetDecimals = IERC20Detailed(input.underlyingAsset).decimals();
    require(underlyingAssetDecimals > 5, Errors.INVALID_DECIMALS);
    ...
    emit ReserveInitialized(
        input.underlyingAsset,
        aTokenProxyAddress,
        address(0),
        variableDebtTokenProxyAddress,
        input.interestRateStrategyAddress
    );
}
```

Remediation Replace frequent require statements with revert and custom errors to improve gas efficiency.

4 Appendix

4.1 About AstraSec

AstraSec is a blockchain security company that serves to provide high-quality auditing services for blockchain-based protocols. With a team of blockchain specialists, AstraSec maintains a strong commitment to excellence and client satisfaction. The audit team members have extensive audit experience for various famous DeFi projects. AstraSec's comprehensive approach and deep blockchain understanding make it a trusted partner for the clients.

4.2 Disclaimer

4.3 Contact

Phone	+86 156 0639 2692
Email	contact@astrasec.ai
Twitter	https://x.com/AstraSecAI