

Scanned Code Report

Code Info

[Developer Scan](#)

 Scan ID	3	 Date	February 15, 2026
 Organization	RigoBlock	 Repository	v3-contracts
 Branch	development	 Commit Hash	39db51e9...bb3f278e

Contracts in scope

[contracts/protocol/extensions/adapters/AUniswapRouter.sol](#)[contracts/protocol/extensions/adapters/interfaces/IUniswapRouter.sol](#)[contracts/protocol/core/state/MixinPoolState.sol](#)

Code Statistics

 Findings	3	 Contracts Scanned	3	 Lines of Code	541
--	---	---	---	---	-----

Findings Summary



-  High Risk (0)
-  Medium Risk (1)
-  Low Risk (1)
-  Info (0)
-  Best Practices (1)

Code Summary

The protocol provides a specialized adapter, `AUniswapRouter`, designed to bridge a Rigoblock smart pool with the Uniswap v4 ecosystem. This integration enables smart pools to perform decentralized trading and manage concentrated liquidity positions directly on Uniswap v4.

The architecture is modular, where the core smart pool contract utilizes this adapter via `delegatecall` to execute complex DeFi operations. The `AUniswapRouter` contract is responsible for securely handling all interactions with Uniswap's Universal Router and Position Manager. It ensures proper validation and safety by managing token approvals through `permit2`, verifying that recipients of swaps are the pool itself, and confirming that tokens received from trades have an established price feed.

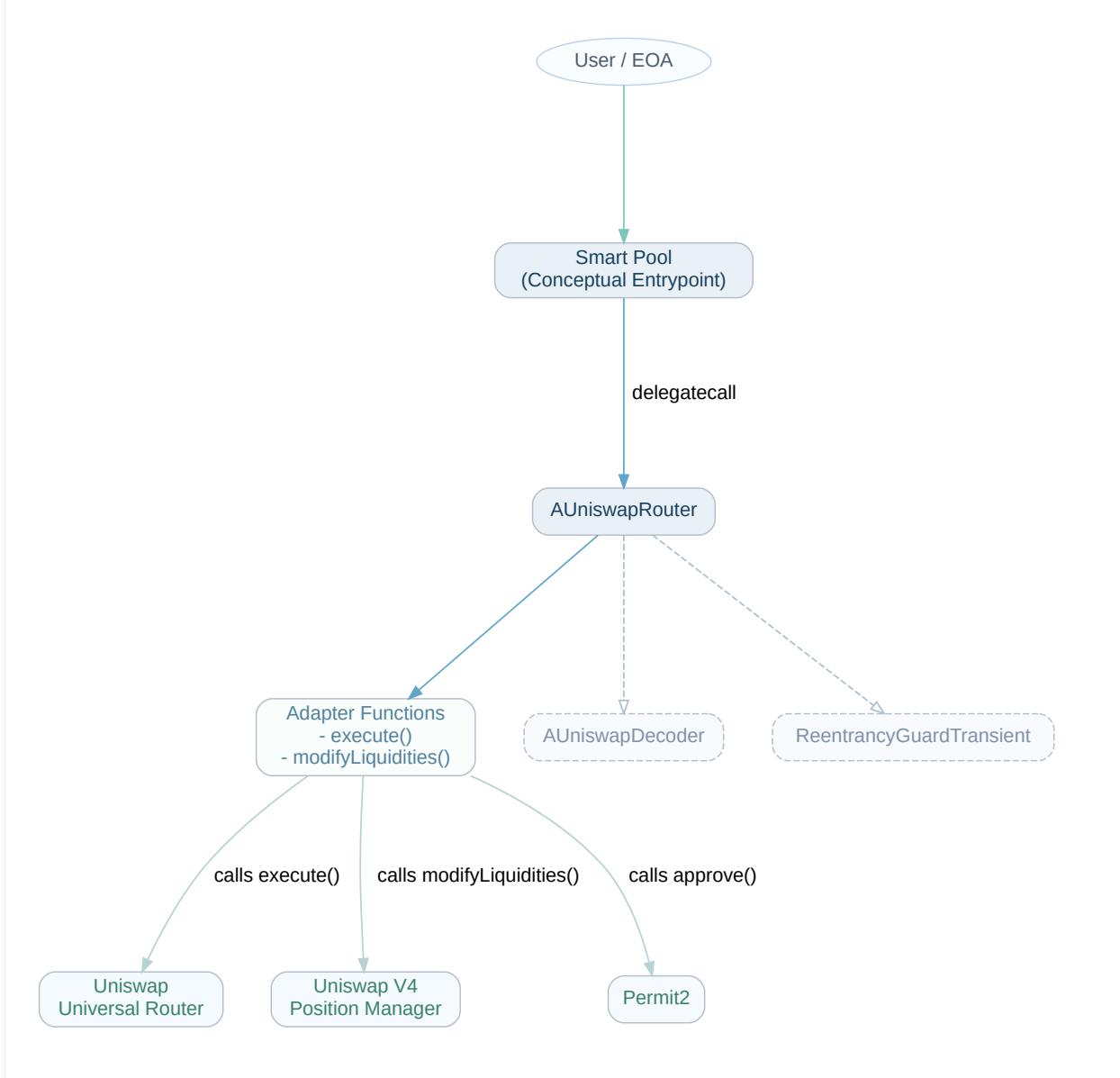
For liquidity management, the adapter allows the pool to mint, modify, and burn Uniswap v4 liquidity positions. It meticulously tracks the pool's NFT position token IDs in storage, ensuring the pool maintains ownership and control over its liquidity. The contract incorporates several security measures, including reentrancy guards, transaction deadline enforcement, and restrictions against direct external calls. It also includes a safeguard against potentially malicious Uniswap hooks that could manipulate liquidity reporting.

Entry Points and Actors

The primary actor is the **Pool Manager/User**, who interacts with the smart pool, which in turn calls the adapter.

- `execute(commands, inputs, deadline)`: Allows the pool to execute a series of arbitrary trading commands on the Uniswap Universal Router, such as token swaps.
- `modifyLiquidities(unlockData, deadline)`: Enables the pool to manage its Uniswap v4 concentrated liquidity positions, including minting new positions, adding or removing liquidity, and burning existing positions.

Code Diagram



 1 of 3 Findings contracts/protocol/extensions/adapters/AUniswapRouter.sol**Storage corruption in BURN_POSITION handling corrupts position tracking**

The `_processTokenIds` function contains a critical bug when handling the BURN_POSITION action that corrupts the position tracking storage. When a position is burned and needs to be removed from the `tokenId` array, the code incorrectly stores the array index instead of the tokenId value, and updates the wrong entry in the positions mapping.

Vulnerable code:

```
if (positions[i].action == Actions.BURN_POSITION) {
    uint256 position = idsSlot.positions[positions[i].tokenId];
    uint256 idIndex = position - 1;
    uint256 lastIndex = idsSlot.tokenIds.length - 1;

    if (idIndex != lastIndex) {
        idsSlot.tokenIds[idIndex] = lastIndex; // BUG: assigns index instead of tokenId
        idsSlot.positions[lastIndex] = position; // BUG: updates wrong mapping key
    }

    idsSlot.positions[positions[i].tokenId] = 0;
    idsSlot.tokenIds.pop();
```

The issue occurs when `idIndex != lastIndex` (the burned position is not the last one in the array). The code attempts to move the last tokenId into the burned position's slot using a swap-and-pop pattern, but makes two critical mistakes:

1. `idsSlot.tokenIds[idIndex] = lastIndex;` - This assigns the array INDEX to the array element, when it should assign the tokenId VALUE at that index:
`idsSlot.tokenIds[idIndex] = idsSlot.tokenIds[lastIndex];`
2. `idsSlot.positions[lastIndex] = position;` - This updates the positions mapping using the array index as the key, when it should use the tokenId being moved: `idsSlot.positions[lastTokenId] = position;`

Impact:

- The `tokenId` array becomes corrupted, containing array indices (small numbers like 0, 1, 2) instead of actual Uniswap V4 NFT token IDs
- The `positions` mapping becomes inconsistent, with incorrect key-value pairs
- Breaks the critical invariant: "Uniswap V4 positions stored in TokenIdsSlot are always in bijection with `idsSlot.tokenIds` array (no duplicates, mapping index != 0)"
- Subsequent operations on positions will fail or behave unpredictably as the storage no longer correctly represents the pool's Uniswap positions
- Could lead to permanent DoS of position management functionality as the storage becomes increasingly corrupted with each burn operation
- May prevent the pool from properly tracking which positions it owns, potentially leading to loss of liquidity access

Example scenario:

- Initial state: `tokenId`s = [100, 200, 300], `positions[100]`=1, `positions[200]`=2, `positions[300]`=3

- Burn tokenId 100 (idIndex=0, lastIndex=2)
- After bug: tokenIds = [2, 200], positions[2]=1, positions[200]=2, positions[300]=0
- The array now contains the number 2 (an index) instead of tokenId 300
- positions[2] incorrectly maps index 2 to array position 1
- positions[300] is cleared but 300 is no longer in the tokenIds array

Severity Note:

- The pool (not end users) controls modifyLiquidity and routinely performs burns as part of rebalancing.
- The pool cannot easily bypass the adapter's tracking to manage/burn NFTs directly in the POSM (no separate emergency operator flow).
- No separate maintenance function exists to re-sync or rebuild idsSlot.tokenIds and idsSlot.positions.

 2 of 3 Findings contracts/protocol/extensions/adapters/AUniswapRouter.sol**modifyLiquidities swallows PositionManager reverts, allowing partial state changes (token activation/approvals) to persist while Uniswap action fails** • Low Risk

`modifyLiquidities` wraps the external call to `_uniV4Posm.modifyLiquidities(...)` in a `try/catch`, but in the generic `catch` branch it only reverts when it detects an insufficient native balance and otherwise returns successfully without bubbling up the underlying failure:

```
try _uniV4Posm.modifyLiquidities{value: newParams.value}(unlockData, deadline) {
    _processTokenIds(...);
    return;
} catch Error(string memory reason) {
    revert(reason);
} catch {
    if (newParams.value > address(this).balance) {
        revert InsufficientNativeBalance();
    }
}
```

Because the generic `catch { ... }` does not revert for most failures, the function can complete successfully even though the PositionManager operation reverted.

This is particularly problematic because `modifyLiquidities` performs externalized and/or persistent side effects *before* calling the PositionManager:

- `_assertTokensOutHavePriceFeed(newParams.tokensOut);` mutates the pool's `activeTokensSet()` by adding tokens.
- `_safeApproveTokensIn(newParams.tokensIn, address(_uniV4Posm));` can mutate ERC20 allowances (pool -> Permit2) and Permit2 allowances (pool -> spender).

If the PositionManager call fails in a way that lands in the generic catch clause (i.e., not a Solidity `Error(string)`), those pre-call changes persist while the intended Uniswap liquidity action does not occur.

Impact:

- The pool can end up with state changes that are inconsistent with the user's expectation of atomicity (e.g., new tokens marked active, approvals updated) despite no Uniswap v4 position changes.
- If `tokensOut` can be influenced via crafted/invalid Uniswap v4 parameters that revert in the PositionManager, this can be used to grow the pool's active token set without completing any Uniswap action, which can increase gas costs and complicate downstream logic that iterates over active tokens.

Severity Note:

- Only the pool (via delegatecall) can invoke `modifyLiquidities`; end users cannot call it directly.
- Permit2 per-spender approvals with expiration=0 do not grant lasting spend rights; only the ERC20->Permit2 allowance persists.
- Active token additions require tokens with valid oracles, limiting arbitrary bloat.

 3 of 3 Findings contracts/protocol/extensions/adapters/AUniswapRouter.sol

Error handling in external calls can be improved

• Best Practices

The `execute` and `modifyLiquidities` functions use `try/catch` blocks to handle potential failures in external calls to the Uniswap router and position manager. However, the `catch` blocks do not effectively handle or propagate error information.

In the `execute` function, the generic `catch (bytes memory returnData)` block attempts to cast the returned data to a string. This will fail if the external contract reverts with a custom error, causing the `AUniswapRouter` to revert with a generic, uninformative error.

In the `modifyLiquidities` function, the generic `catch` block does not attempt to use the `returnData` at all, meaning any revert reason from the Position Manager is completely swallowed.

```
// in execute()
} catch (bytes memory returnData) {
    if (params.value > address(this).balance) {
        revert InsufficientNativeBalance();
    } else {
        revert(string(returnData)); // This can fail
    }
}

// in modifyLiquidities()
} catch Error(string memory reason) {
    revert(reason);
} catch { // Swallows any custom error from the external call
    if (newParams.value > address(this).balance) {
        revert InsufficientNativeBalance();
    }
}
```

Additionally, both functions perform a check for `InsufficientNativeBalance` after the external call has already failed. A pre-flight check before the `try` block would be clearer and more robust. While this does not cause a vulnerability, it significantly hinders debugging and off-chain monitoring by obscuring the true reason for a transaction failure.

Disclaimer

Kindly note, no guarantee is being given as to the accuracy and/or completeness of any of the outputs the AuditAgent may generate, including without limitation this Report. The results set out in this Report may not be complete nor inclusive of all vulnerabilities. The AuditAgent is provided on an 'as is' basis, without warranties or conditions of any kind, either express or implied, including without limitation as to the outputs of the code scan and the security of any smart contract verified using the AuditAgent.

Blockchain technology remains under development and is subject to unknown risks and flaws. This Report does not indicate the endorsement of any particular project or team, nor guarantee its security. Neither you nor any third party should rely on this Report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset.

To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this Report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.