Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Developer Scan

Scan ID 1	Date February 13, 2026
Organization RigoBlock	Repository v3-contracts
Branch development	Commit Hash 39db51e9...bb3f278e

Contracts in scope

- contracts/protocol/extensions/adapters/AIntents.sol
- contracts/protocol/extensions/adapters/interfaces/IAIntents.sol
- contracts/protocol/extensions/ECrosschain.sol
- contracts/protocol/extensions/adapters/interfaces/IECrosschain.sol

Code Statistics

Findings	Contracts Scanned	Lines of Code
5	4	500

Findings Summary

Total Findings

- High Risk (0)
- Medium Risk (3)
- Low Risk (1)
- Info (0)
- Best Practices (1)

Code Summary

This protocol provides a cross-chain asset management extension for Rigoblock smart pools, enabling them to bridge tokens between different blockchains using the Across Protocol. The system is designed to maintain the integrity of the pool's Net Asset Value (NAV) during these transfers.

The architecture is split into two main components that operate via `delegatecall` from a smart pool:

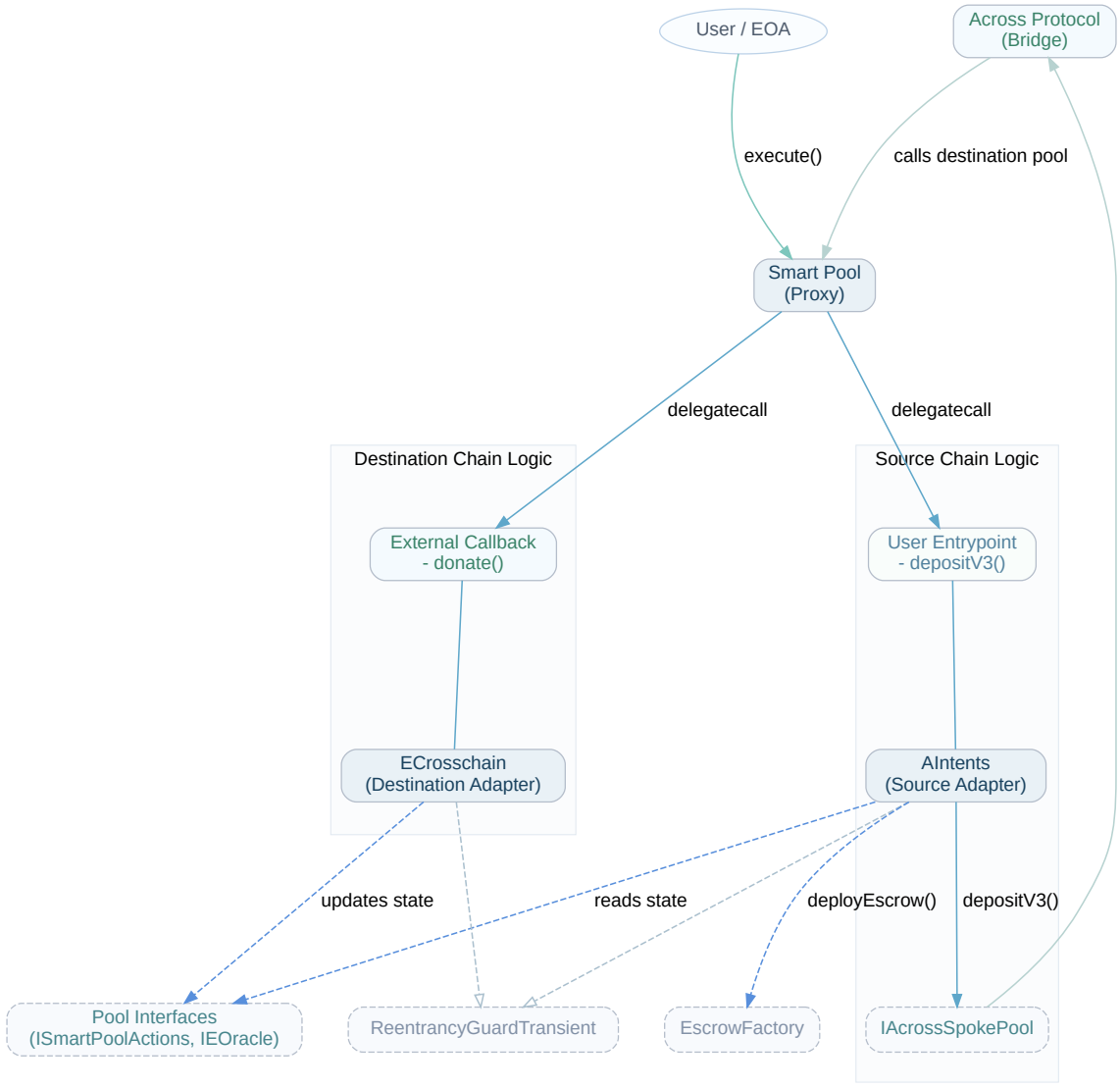
- **AIntents**: This contract acts as the sending adapter. It initiates cross-chain transfers by depositing tokens into the Across Protocol. It supports two distinct operation types: `Transfer`, which adjusts the pool's virtual supply to reflect assets moving out, and `Sync`, which is used for asset rebalancing without affecting user shares. For each transfer, it deploys a temporary escrow contract to securely manage potential refunds from Across.
- **ECrosschain**: This contract serves as the receiving extension. It processes incoming tokens on the destination chain. To prevent NAV manipulation, it employs a two-step locking mechanism that records the pool's state before the token transfer and validates it afterward. When handling a `Transfer` operation, it updates the pool's virtual supply to account for the newly arrived assets, effectively rebalancing the pool's share accounting across chains.

Entry Points and Actors

The main entry points for modifying the protocol's state are:

- `depositV3(AcrossParams calldata params)`: Initiates a cross-chain token transfer from the smart pool. This function is called by a **Pool Operator** or an authorized user.
- `donate(address token, uint256 amount, DestinationMessageParams calldata params)`: Receives tokens on the destination chain to finalize a cross-chain transfer or process a refund. This function is primarily called by the **Across Protocol's multicall handler** or the **escrow contract** created during the deposit.

Code Diagram



1 of 5 Findings

contracts/protocol/extensions/adapters/AIntents.sol

Missing minimum effective supply validation in Transfer mode allows pool drain beyond safe threshold

Medium Risk

The `_handleSourceTransfer` function writes negative virtual supply to the pool when initiating cross-chain transfers in Transfer mode, but does not validate that the resulting effective supply remains above the minimum required threshold of `totalSupply / MINIMUM_SUPPLY_RATIO` (12.5% with MINIMUM_SUPPLY_RATIO = 8).

The vulnerable code in `_handleSourceTransfer`:

```
function _handleSourceTransfer(AcrossParams memory params) private {
    (uint256 outputValueInBase, ) = _getOutputValueInBase(params);
    NetAssetsValue memory navParams = ISmartPoolActions(address(this)).updateUnitaryValue();
    uint8 poolDecimals = StorageLib.pool().decimals;

    int256 burntAmount = ((outputValueInBase * (10 ** poolDecimals)) /
navParams.unitaryValue).toInt256();

    // Write negative VS (shares leaving this chain → reduces effective supply)
    (-burntAmount).updateVirtualSupply();
    // MISSING: No validation that effectiveSupply >= totalSupply / MINIMUM_SUPPLY_RATIO
}
```

The effective supply is calculated as `totalSupply + virtualSupply`. When negative virtual supply is written, it reduces the effective supply. Without validation, an operator could transfer tokens worth 90% of the pool's value in a single transaction, pushing the effective supply below the 12.5% minimum threshold.

This constraint is critical for pool functionality as stated in invariant 7: "Negative virtual supply written on source chain must equal –sharesLeaving and cannot push effectiveSupply below totalSupply/MINIMUM_SUPPLY_RATIO". The documentation explicitly states this 12.5% safety buffer "prevents supply exhaustion" and "ensures pool remains functional with positive effective supply".

Without this check, the pool could end up with an effective supply that violates the minimum ratio, potentially causing:
1. Incorrect NAV calculations (NAV = totalValue / effectiveSupply with very small denominator)
2. Pool becoming non-functional for withdrawals
3. Ability to manipulate per-share value by reducing effective supply
4. Breaking the post-burn protection mechanism that relies on this constraint

Note that Sync mode includes validation via `NavImpactLib.validateNavImpact`, but Transfer mode has no such protection despite modifying virtual supply in a way that could violate the constraint.

Severity Note:
- depositV3/Transfer is callable only through the pool via delegatecall and is effectively restricted to privileged operators/governance.
- Withdrawal/mint/NAV paths depend on effectiveSupply and lack an independent minimum-supply floor in Transfer mode beyond what is shown.

- Operators can restore state by performing offsetting cross-chain ops or sync actions, making the issue recoverable.

2 of 5 Findings

contracts/protocol/extensions/ECrosschain.sol

`shouldUnwrapNative` path mixes WETH temporary balance with native-ETH accounting, causing deterministic NAV-integrity reverts (DoS) whenever there is any pre-existing wrappedNative balance and native was not already active

Medium Risk

`ECrosschain.donate()` records the temporary balance and stores NAV/assets during the `amount == 1` "lock" call using the originally provided `token`.

During the "finalize" call (`amount > 1`), when `token == wrappedNative && params.shouldUnwrapNative` the function unwraps WETH and then *mutates* `token` to `address(0)` (native ETH) **before** computing `previouslyActive` and before NAV-integrity validation:

```
if (token == wrappedNative && params.shouldUnwrapNative) {
    IWETH9(wrappedNative).withdraw(amountDelta);
    token = address(0);
}

bool previouslyActive = StorageLib.isOwnedToken(token);
...
_validateNavIntegrity(token, amountDelta, storedBalance, previouslyActive);
```

However, `storedBalance` was fetched from transient storage using the *pre-unwrapping* token (WETH), i.e. it is the pool's WETH balance recorded during the lock phase:

```
(uint256 storedBalance, bool initialized) = token.getTemporaryBalance();
```

Inside `_validateNavIntegrity`, when `previouslyActive == false` it adds `storedBalance` to `amountDelta`:

```
if (!previouslyActive) {
    amountDelta += storedBalance;
}
```

After unwrapping, `previouslyActive` refers to whether **native ETH (`address(0)`)** was active, but `storedBalance` still represents **WETH** balance at lock time.

This creates a deterministic mismatch in the expected-assets computation whenever the pool had any pre-existing WETH balance at the time of the lock call and native ETH was not already active:
- The stored snapshot (`TransientStorage.getStoredAssets()`) may already include the pre-existing WETH value (if WETH was active), or may exclude it (if WETH was inactive).
- After finalize+unwrap, the pool ends up with (roughly) the same WETH balance as before (the delta got unwrapped) plus additional native ETH.
- The code nevertheless treats the pre-existing **WETH** `storedBalance` as if it were a pre-existing **native ETH** balance and adds it to `amountDelta`, causing `expectedAssets` to be too large by `storedBalance`'s value.

The result is a revert with `NavManipulationDetected(...)` even if no manipulation occurred, which makes any Across fill that uses `shouldUnwrapNative == true` fail on destination under common portfolio states (e.g., pools

that already hold/track WETH, but have not previously activated native ETH). This blocks the intended cross-chain receive flow and forces reliance on refunds/escrow recovery paths instead of successful fills.

Severity Note:
- The pool commonly holds a non-zero WETH balance at lock time.
- Native ETH is not already active in activeTokensSet() when finalize runs.

3 of 5 Findings

contracts/protocol/extensions/ECrosschain.sol

Strict equality check in NAV validation can cause legitimate donations to fail

Medium Risk

The `_validateNavIntegrity` function uses a strict equality check to validate that NAV was not manipulated during the donation process:

```
require(
    navParams.netTotalValue == expectedAssets,
    NavManipulationDetected(expectedAssets, navParams.netTotalValue)
);
```

This strict equality check can cause legitimate donations to fail in several scenarios:

1. **Oracle Price Updates**: If a Chainlink price feed updates between the first `donate(amount=1)` call (which stores NAV/assets) and the second `donate(amount>1)` call (which validates), the NAV calculation will change even though no manipulation occurred. The two calls happen in the same transaction via multicall, but `updateUnitaryValue()` is called in both, and oracle prices can update within a single transaction if the oracle's `latestRoundData()` returns a new round.

2. **Yield-Bearing Tokens**: If the pool holds yield-bearing tokens (like stETH, aTokens, or other rebasing tokens), the balance can increase between the two calls, causing the NAV to increase legitimately.

3. **Rounding Errors**: Token conversions through `convertTokenAmount()` involve division operations that round down. When converting the same amount through different paths (once during storage, once during validation), small rounding differences can cause inequality.

For example, if a pool holds a yield-bearing token:
- First call stores: `storedAssets = 1000000000000000000` (current NAV)
- Yield accrues: pool balance increases by 1 wei
- Second call validates: `navParams.netTotalValue = 1000000000000000001`
- Validation fails even though donation was legitimate

This creates a denial-of-service condition where valid cross-chain transfers can fail unpredictably.

4 of 5 Findings

contracts/protocol/extensions/adapters/AIntents.sol

`depositV3` forwards `sourceNativeAmount` from the pool without verifying `msg.value`, allowing callers to spend the pool's native balance (ETH) even when they did not supply ETH

Low Risk

`AIntents.depositV3()` decodes `SourceMessageParams` and then forwards `sourceParams.sourceNativeAmount` as call value to the Across SpokePool:

```
_acrossSpokePool.depositV3{value: sourceParams.sourceNativeAmount}(
    ...
);
```

There is no check that the external caller actually supplied that ETH as `msg.value`.

Because this adapter is intended to run via `delegatecall` from the pool proxy, the ETH sent in the Across deposit is ultimately taken from the pool's balance. If `sourceParams.sourceNativeAmount > 0` and the pool already has native ETH (and has it marked active, per the existing `StorageLib.isOwnedToken(address(0))` check), then:
- A caller can set a non-zero `sourceNativeAmount` while sending `msg.value == 0`.
- The call can still attempt to send ETH out of the pool to `SpokePool.depositV3`.

This breaks the "value forwarding" invariant and creates a permission-surface issue: any account that is allowed to call `depositV3` (e.g., "authorized users" per the project description) may be able to force the pool to spend its pre-existing native ETH balance as part of a bridge deposit, even if that account did not fund the call with ETH and even if the intended authorization model was to require callers to pay the native component themselves.

Severity Note:
- Authorized users can trigger the pool to delegatecall this adapter's depositV3.
- Across SpokePool accepts native value when inputToken == wrappedNative, so the forwarded ETH leaves the pool.
- Pools may hold native ETH and have it marked active in some configurations.

5 of 5 Findings

contracts/protocol/extensions/ECrosschain.sol

NavImpactLib validation missing for virtual supply updates on destination chain • Best Practices

In Transfer mode on the destination chain, the `_updateVirtualSupply` function writes positive virtual supply without validating that the effective supply stays within safe bounds:

```
function _updateVirtualSupply(address token, uint256 amount) private {
    address baseToken = StorageLib.pool().baseToken;
    uint256 amountValueInBase = IEOracle(address(this))
        .convertTokenAmount(token, amount.toInt256(), baseToken)
        .toUint256();

    uint8 poolDecimals = StorageLib.pool().decimals;
    uint256 storedNav = TransientStorage.getStoredNav();
    uint256 mintedAmount = ((amountValueInBase * (10 ** poolDecimals)) / storedNav);

    mintedAmount.toInt256().updateVirtualSupply();
}
```

While the source chain validates minimum effective supply (or should, per previous finding), the destination chain does not validate that positive virtual supply doesn't create issues. For instance:

- If multiple large transfers arrive concurrently to a destination chain with low total supply
- Each writes large positive VS
- Effective supply could balloon to many times the total supply
- NAV calculations become unstable

While positive VS is less dangerous than negative VS (it doesn't prevent withdrawals), extremely high VS ratios could cause:

1. Integer overflow in effective supply calculations
2. Precision loss in NAV calculations
3. Unexpected behavior in pool operations that depend on supply ratios

The destination chain should validate that effective supply remains within reasonable bounds, for example:

```
int256 currentVS = VirtualStorageLib.getVirtualSupply();
int256 newVS = currentVS + mintedAmount.toInt256();
uint256 totalSupply = IERC20(address(this)).totalSupply();

require(
    newVS <= totalSupply.toInt256() * 10,
    EffectiveSupplyTooHigh()
);
```

This would prevent the effective supply from exceeding 10x the total supply, maintaining stability in NAV calculations.

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.