Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Developer Scan

Scan ID 2	Date February 15, 2026
Organization RigoBlock	Repository v3-contracts
Branch development	Commit Hash 39db51e9...bb3f278e

Contracts in scope

contracts/protocol/core/actions/MixinActions.sol

contracts/protocol/core/state/MixinPoolValue.sol

Code Statistics

Findings 2	Contracts Scanned 2	Lines of Code 492

Findings Summary

Total Findings

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

Code Summary

The protocol implements a decentralized asset management system, often referred to as a "smart pool" or vault. It allows users to collectively invest in a managed portfolio of crypto assets. The core functionality revolves around minting and burning pool shares, which represent a user's stake in the pool's total assets.

The pool's value, or Net Asset Value (NAV), is dynamically calculated based on the collective worth of all assets it holds. This valuation process is comprehensive, accounting not only for the tokens held directly by the contract but also for assets deployed in external DeFi applications and protocols. An integrated oracle is crucial for this process, as it converts the value of all diverse assets into a single `baseToken` denomination to determine the accurate NAV per share.

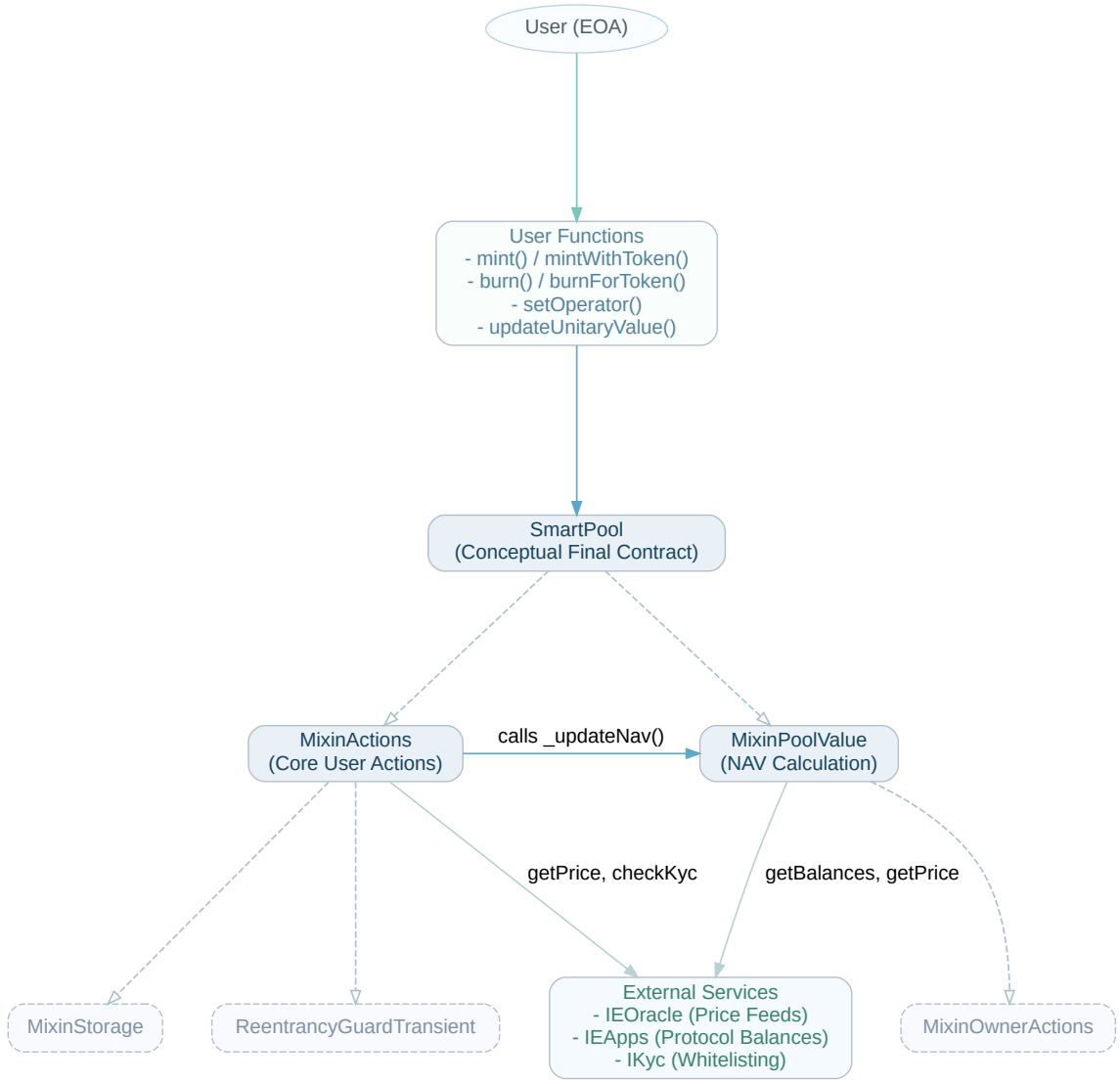
Key features of the protocol include:
- **Multi-Asset Deposits**: Users can mint pool shares by depositing either the pool's designated `baseToken` or other whitelisted ERC20 tokens.
- **Flexible Withdrawals**: Users can burn their shares to redeem the underlying assets, primarily in the `baseToken`. A fallback mechanism also allows redemption in other active tokens if the pool's `baseToken` liquidity is low.
- **External Application Integration**: The pool is designed to interact with other DeFi protocols, allowing it to deploy its assets for strategies like yield farming or liquidity provision.
- **Fee Structure**: The protocol incorporates a dual-fee system on mint and burn operations, consisting of a spread fee and a transaction fee, which contribute to the pool's treasury or management.
- **Investor Timelocks**: A configurable minimum holding period can be enforced, requiring investors to lock their shares for a certain duration after minting.
- **Operator Delegation**: Users can authorize an "operator" address to manage their position on their behalf, enabling more complex management strategies.
- **Optional KYC**: The system can be configured to enforce Know Your Customer (KYC) checks, restricting participation to whitelisted users.

Entry Points & Actors

The primary actors are **Investors**, who deposit and withdraw assets, and **Operators**, who can be delegated to manage an Investor's position.

- `mint(recipient, amountIn, amountOutMin)`: An Investor deposits the pool's base token to mint new pool shares.
- `mintWithToken(recipient, amountIn, amountOutMin, tokenIn)`: An Investor deposits an accepted ERC20 token to mint new pool shares.
- `burn(amountIn, amountOutMin)`: An Investor burns their pool shares to redeem their portion of the underlying assets in the base token.
- `burnForToken(amountIn, amountOutMin, tokenOut)`: An Investor burns their pool shares to redeem assets in a specified alternative token.
- `updateUnitaryValue()`: Any user can call this function to trigger an on-demand recalculation of the pool's Net Asset Value (NAV).
- `setOperator(operator, approved)`: An Investor grants or revokes permission for an Operator to manage their shares.

Code Diagram



1 of 2 Findings

contracts/protocol/core/state/MixinPoolValue.sol

contracts/protocol/core/actions/MixinActions.sol

First NAV initialization sets unitaryValue to 10**decimals regardless of existing assets/virtual supply, allowing value capture by the first minter when the pool is pre-funded

Medium Risk

When `poolTokens().unitaryValue` is zero, `_updateNav()` initializes NAV to a fixed default value (`10**decimals`) and explicitly skips computing NAV from current assets:

"// first mint skips nav calculation\nif (components.unitaryValue == 0) {\n components.unitaryValue = 10 ** components.decimals;\n} else {\n ... compute NAV from assets/effective supply ...\n}".

However, `_updateNav()` still computes the current pool assets via `_computeTotalPoolValue()` *before* that branch:
"int256 netValue = _computeTotalPoolValue(components.baseToken);\n... components.netTotalValue = uint256(netValue);".

As a result, any assets already held by the pool prior to the first mint (e.g., direct transfers to the pool address, accidental transfers, or assets arriving from external mechanisms before the first mint on that chain) do not influence the initial share price.

Because `_mint()` prices new shares using `components.unitaryValue`:
"uint256 mintedAmount = (amountIn * 10 ** components.decimals) / components.unitaryValue;"

the first minter can acquire shares at the fixed default price even if `components.netTotalValue` is already non-zero. After that first mint sets a non-zero `totalSupply`, subsequent NAV updates will incorporate the previously-held assets, increasing `unitaryValue`. The first minter can then burn their shares to redeem a pro-rata portion of those pre-existing assets.

This is an exploitable value-capture mechanism whenever the pool can become pre-funded before the first mint (including scenarios where value is present due to integration flows rather than a normal mint).

Severity Note:
- The pool can receive base-token assets (ERC20 or via integrations) before the first mint.
- If KYC is enabled, the attacker can be whitelisted or an insider could exploit; otherwise minting is public.
- The attacker can wait out the minimum holding period before burning.
- No pause/gate prevents first public mint with base token.

2 of 2 Findings

contracts/protocol/core/actions/MixinActions.sol

No invariant enforcement that `transactionFee > 0` implies a non-zero fee collector; fees can be credited to the zero address and become irrecoverable

Info

Both `_allocateMintTokens(...)` and `_allocateBurnTokens(...)` credit pool-token fees to the address returned by `_getFeeCollector()` whenever `poolParams().transactionFee != 0`. There is no validation in `MixinActions` that the fee collector is non-zero.

Relevant code in mint path:

```
if (transactionFee != 0) {
    address feeCollector = _getFeeCollector();

    if (feeCollector != recipient) {
        uint256 feePool = (mintedAmount * transactionFee) / _FEE_BASE;
        mintedAmount -= feePool;

        accounts().userAccounts[feeCollector].userBalance += uint208(feePool);
        accounts().userAccounts[feeCollector].activation = activation;
        emit Transfer(_ZERO_ADDRESS, feeCollector, feePool);
    }
}
```

And in burn path:

```
if (poolParams().transactionFee != 0) {
    address feeCollector = _getFeeCollector();

    if (msg.sender != feeCollector) {
        uint256 feePool = (amountIn * poolParams().transactionFee) / _FEE_BASE;
        amountIn -= feePool;

        accounts().userAccounts[feeCollector].userBalance += uint208(feePool);
        accounts().userAccounts[feeCollector].activation = uint48(block.timestamp + 1);
        emit Transfer(msg.sender, feeCollector, feePool);
    }
}
```

If `_getFeeCollector()` returns `address(0)` while `transactionFee` is non-zero, fee shares are assigned to the zero address. Since the zero address cannot initiate burns, these fee shares can become permanently stuck, and fee accounting can diverge from the intended operational model (fees being withdrawable/usable by a real collector address).

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.