Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Developer Scan

Scan ID 1	Date February 21, 2026
Organization hakolabs	Repository hako-hardhat
Branch main	Commit Hash 5e4d4473...fabba9cf

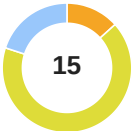
Contracts in scope

- contracts/HakoProxy.sol
- contracts/HakoStableGateway.sol
- contracts/HakoStableVault.sol
- contracts/stable/HakoStableGatewayStorage.sol
- contracts/stable/HakoStableGatewayTypes.sol
- contracts/stable/HakoStableVaultRegistry.sol
- contracts/stable/HakoStableVaultStorage.sol
- contracts/stable/HakoStableVaultTypes.sol
- contracts/shared/VaultErrors.sol
- contracts/shared/VaultEvents.sol
- contracts/shared/VaultMath.sol
- contracts/shared/VaultTypes.sol

Code Statistics

Findings 15	Contracts Scanned 12	Lines of Code 1853

Findings Summary



Total Findings

- High Risk (0)
- Medium Risk (2)
- Low Risk (10)
- Info (3)
- Best Practices (0)

Code Summary

The Hako protocol is a cross-chain stablecoin vault system designed to aggregate deposits from multiple blockchains and generate yield. The architecture is composed of a central `HakoStableVault` on a home chain and multiple `HakoStableGateway` contracts on remote chains.

The `HakoStableVault` serves as the core accounting layer and issues ERC20 LP shares to depositors. It tracks the total value of all assets managed by the protocol across all connected chains. The `HakoStableGateway` contracts act as custody and execution layers on remote chains, handling local deposits and processing withdrawal payouts.

Users can deposit allowlisted stablecoins into the system from any supported chain. Deposits made on the home chain are handled directly by the `HakoStableVault`, while deposits on remote chains are processed by the corresponding `HakoStableGateway`. All deposit amounts are normalized to a common 18-decimal precision for internal accounting. The protocol also supports non-EVM users by creating deterministic pseudo-addresses on the home chain to hold their LP shares.

To generate yield, assets held within the vault and gateways can be allocated to external, allowlisted ERC-4626 compliant vaults. An `Asset Manager` role is responsible for managing these investments. Profits and losses from these strategies are reported back to the `HakoStableVault`, which adjusts the total managed assets and, consequently, the value of the LP shares.

The protocol implements a performance fee on the generated profits, which is calculated using a high-water-mark mechanism. This ensures that fees are only levied on new profits, protecting users from paying fees on recovered losses.

Withdrawals are initiated through the `HakoStableVault`, where users can request to redeem their LP shares for a specific stablecoin on any supported destination chain. This process locks the user's LP shares. A `Withdraw Finalizer` role then completes the request, triggering the asset payout on the target chain and burning the locked shares on the home chain.

Entry Points and Actors

The protocol exposes several functions for different actors:

HakoStableVault

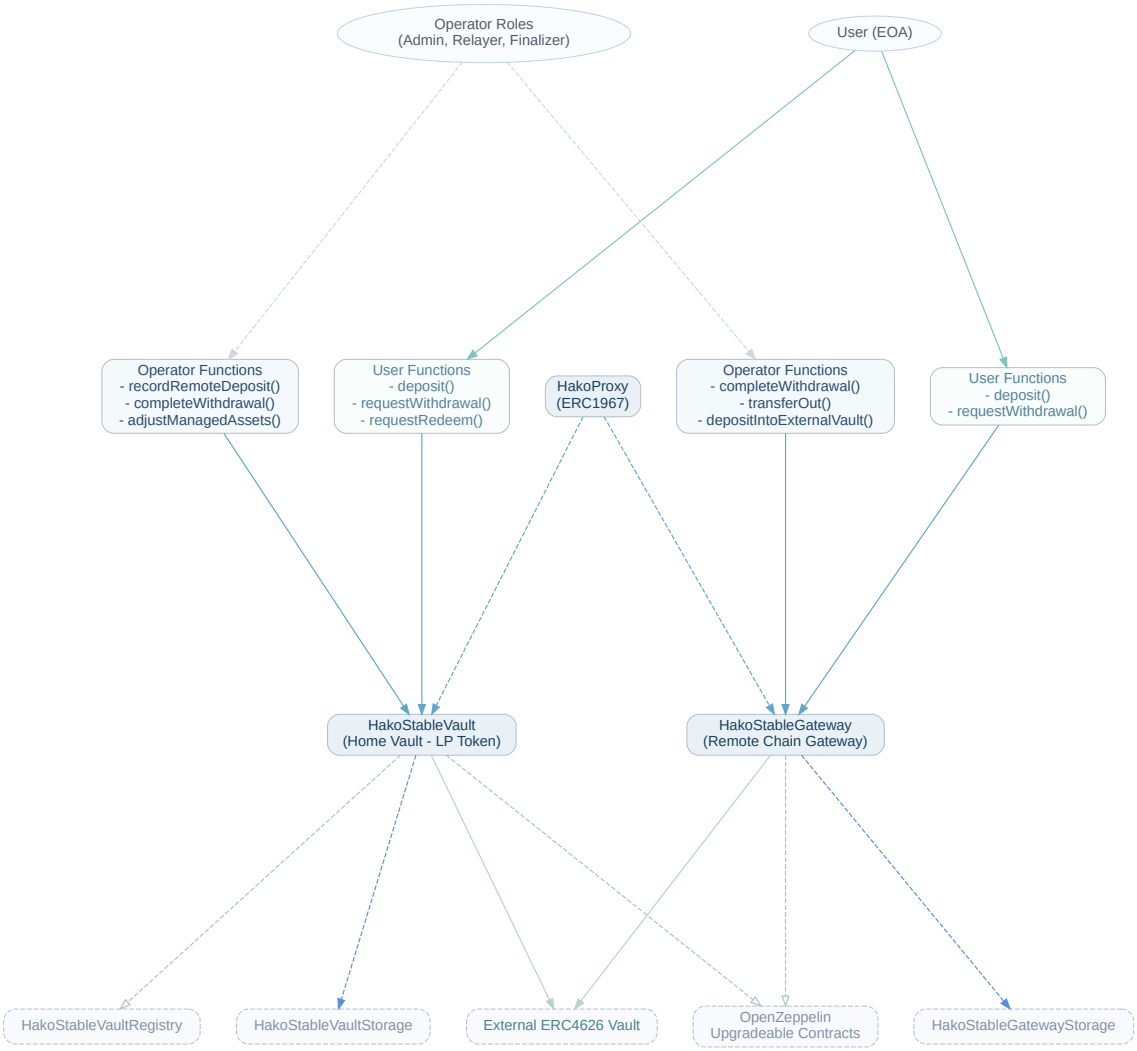
- **User/Depositor**
  - `deposit`: Deposits an allowlisted local token and mints vault LP shares.
  - `requestWithdrawal`: Creates a withdrawal request to a destination chain by specifying a target asset amount, which locks the corresponding shares.
  - `requestRedeem`: Creates a withdrawal request by specifying a fixed number of shares to redeem.
- **Relayer**
  - `recordRemoteDeposit`: Records a deposit from a remote gateway and mints LP shares for an EVM receiver.
  - `recordRemoteDepositNonEvm`: Records a remote deposit and mints LP shares for a non-EVM receiver's pseudo-address.
  - `requestWithdrawalController`: Creates a withdrawal request on behalf of a user, with nonce protection.
  - `recordRemoteWithdrawalRequest`: Records a withdrawal request initiated on a remote gateway.

- **Withdraw Finalizer**
  - `completeWithdrawal`: Finalizes a pending withdrawal, burning the user's locked shares.
  - `cancelWithdrawal`: Cancels a pending withdrawal, unlocking the user's shares.
- **Asset Manager**
  - `transferOut`: Transfers assets out of the vault for bridging or operational purposes.
  - `adjustManagedAssets`: Adjusts the total managed assets based on reported profits or losses from strategies.
  - `depositIntoExternalVault`: Deposits assets into an external ERC-4626 yield vault.
  - `withdrawFromExternalVault`: Withdraws assets from an external ERC-4626 vault.
  - `redeemFromExternalVault`: Redeems shares from an external ERC-4626 vault for underlying assets.

HakoStableGateway

- **User/Depositor**
  - `deposit`: Deposits an allowlisted stablecoin into the remote gateway.
  - `requestWithdrawal`: Creates a withdrawal request for a payout on the local gateway chain.
- **Withdraw Finalizer**
  - `completeWithdrawal`: Completes a pending withdrawal request and pays out the token to the receiver.
  - `cancelWithdrawal`: Cancels a pending withdrawal request.
- **Asset Manager**
  - `transferOut`: Transfers assets out of the gateway for bridging or operational purposes.
  - `depositIntoExternalVault`: Deposits assets into an external ERC-4626 yield vault.
  - `withdrawFromExternalVault`: Withdraws assets from an external ERC-4626 vault.
  - `redeemFromExternalVault`: Redeems shares from an external ERC-4626 vault for underlying assets.

Code Diagram



1 of 15 Findings

contracts/HakoStableVault.sol

Pending withdrawals are not subject to losses, unfairly penalizing remaining LPs

Medium Risk

When a user requests a withdrawal, the normalized asset amount to be paid out (`amountNormalized`) is calculated and stored. This value is fixed at the time of the request. If the vault subsequently realizes a loss (reported via `adjustManagedAssets`) before the withdrawal is finalized, the share price drops. However, when the withdrawal is completed via `completeWithdrawal`, the user still receives the original, higher `amountNormalized`. The contract reduces `totalManagedAssets` by this fixed amount, which is greater than the current value of the shares being burned. This discrepancy causes the loss to be disproportionately borne by the remaining liquidity providers, who effectively subsidize the exiting user's withdrawal.

This creates a critical vulnerability where informed users can front-run the reporting of a major loss (e.g., after a hack of an external strategy) by submitting a withdrawal request. This allows them to exit at a pre-loss valuation, magnifying the financial damage for the LPs who remain in the vault.

Example Scenario:
1. Initial state: `totalManagedAssets` = 10,000, `totalSupply` = 10,000. Price per share = 1.
2. A user with 1,000 shares calls `requestWithdrawal` for 1,000 `amountNormalized`. The request is stored, and 1,000 shares are locked.
3. An asset manager reports a 5,000 asset loss. The `adjustManagedAssets` function is called, and `totalManagedAssets` becomes 5,000. The share price drops to 0.5.
4. A finalizer calls `completeWithdrawal` for the user's pending request.

```
// HakoStableVault.sol:completeWithdrawal
function completeWithdrawal(uint256 requestId) external onlyRole(WITHDRAW_FINALIZER_ROLE)
nonReentrant {
    HakoStableVaultStorage.Layout storage layout_ = HakoStableVaultStorage.layout();
    WithdrawalRequest storage req = layout_.withdrawalRequests[requestId];
    if (req.status != WithdrawalStatus.Pending) revert WithdrawalNotPending(requestId);

    req.status = WithdrawalStatus.Completed;

    layout_.lockedShares[req.owner] -= req.sharesLocked;
    if (layout_.totalManagedAssets < req.amountNormalized) revert ManagedAssetsUnderflow();
    layout_.totalManagedAssets -= req.amountNormalized; // Subtracts the original 1,000

    _burn(req.owner, req.sharesLocked); // Burns 1,000 shares (now worth only 500)

    emit WithdrawalCompleted(requestId, req.owner, req.sharesLocked, req.amountNormalized);
}
```

In this scenario, `totalManagedAssets` is reduced from 5,000 to 4,000, while `totalSupply` is reduced from 10,000 to 9,000. The new share price becomes `4000 / 9000 = 0.444`. The remaining LPs' share value dropped from 0.5 to 0.444 because they covered the 500 asset difference for the exiting user.

Severity Note:
- Operators will proceed to finalize existing pending withdrawals post-loss rather than cancel all of them.

- Off-chain/operational settlement honors the stored amountNormalized for payment.
- There exists a window between incident knowledge and adjustManagedAssets where LPs can submit requests.

2 of 15 Findings contracts/HakoStableVault.sol contracts/HakoStableGateway.sol

Risky Strict Equality Check • Medium Risk

Identifies risky use of strict equality checks (==) on complex data types that may have precision issues.

HakoStableVault.sol instances:
- `shareBalance == 0` at line 632 in `getExternalVaultPositions()` (lines 621-657)
- `shareBalance_scope_3 == 0` at line 645 in `getExternalVaultPositions()` (lines 621-657)

HakoStableGateway.sol instances:
- `shareBalance == 0` at line 338 in `getExternalVaultPositions()` (lines 326-363)
- `shareBalance_scope_3 == 0` at line 351 in `getExternalVaultPositions()` (lines 326-363)

Strict equality checks on balance values can fail unexpectedly due to rounding or precision loss, potentially causing logic errors or unexpected behavior in vault position calculations.

3 of 15 Findings contracts/HakoStableVault.sol contracts/shared/VaultMath.sol

First depositor can capture all managed assets if totalManagedAssets > 0 while totalSupply == 0 (share-minting initializes 1:1)

Low Risk

The vault's share issuance logic mints shares 1:1 with normalized assets whenever either `totalSupply()` or `totalManagedAssets` is zero. This is implemented in `VaultMath.previewMintShares` and used by `_mintSharesForDeposit`.

Relevant code:
"""solidity
function previewMintShares(uint256 amountNormalized, uint256 supply, uint256 managed) internal pure returns (uint256) {
if (supply == 0 || managed == 0) return amountNormalized;
return (amountNormalized * supply) / managed;
}

function *mintSharesForDeposit(address receiver, uint256 amountNormalized) internal returns (uint256 sharesMinted) {*
...
uint256 supply = totalSupply();
uint256 managed = layout.totalManagedAssets;

```
sharesMinted = VaultMath.previewMintShares(amountNormalized, supply, managed);
...
layout_.totalManagedAssets = managed + amountNormalized;
_mint(receiver, sharesMinted);
```

}
"""

If the system ever reaches a state where `layout_.totalManagedAssets > 0` while `totalSupply() == 0`, the next depositor receives shares as if the vault had no pre-existing assets (shares minted equal `amountNormalized`). Immediately after, that depositor becomes the sole (or dominant) share holder while `totalManagedAssets` includes both the pre-existing managed assets and their small deposit.

This creates an externally exploitable "initialization capture" condition:
1) The vault has `totalSupply() == 0` but `totalManagedAssets == M` (non-zero).
2) An attacker deposits the minimum allowed amount `d`.
3) Attacker receives `sharesMinted = d` (1:1), and the vault becomes `totalManagedAssets = M + d`, `totalSupply = d`.
4) The attacker can then redeem/burn their shares to withdraw essentially `M + d` normalized assets' worth via the withdrawal request flow.

A `totalManagedAssets > 0` / `totalSupply == 0` state can arise from privileged accounting actions (e.g., `adjustManagedAssets(+delta)` being called before any shares exist), or from operational/flow mistakes that burn the last remaining shares while leaving `totalManagedAssets` non-zero (e.g., burning all shares via an amount-based request that deducts less than all managed assets). The impact is that pre-existing managed value becomes claimable by a fresh depositor who contributed only the minimum deposit amount.

Severity Note:
- No malicious/erroneous privileged action seeds a large managed balance while totalSupply == 0; only rounding-related remainders are considered for realistic impact.

4 of 15 Findings

contracts/HakoStableVault.sol

Performance fee calculation uses approximation that slightly underpays protocol

Low Risk

The performance fee calculation in `VaultMath.calculatePerformanceFeeShares` uses an approximate formula for computing fee shares:

```
uint256 feeAmount = (taxableProfit * performanceFeeBps) / 10_000;
if (feeAmount == 0) {
    return (0, currentPrice, taxableProfit);
}
feeShares = (feeAmount * supply) / managed;
```

Where `managed` includes the profit and `supply` is before minting fee shares. This calculates:

```
feeShares = feeAmount * S / (M + P)
```

After minting fee shares, the value of those shares becomes:

```
feeValue = feeShares * (M + P) / (S + feeShares)
```

For the fee to be exact, we need:

```
feeShares * (M + P) / (S + feeShares) = feeAmount
```

Solving for feeShares:

```
feeShares = feeAmount * S / (M + P - feeAmount)
```

The code uses `(M + P)` in the denominator instead of `(M + P - feeAmount)`, resulting in slightly fewer fee shares being minted than mathematically correct. For typical performance fees (e.g., 20%) on modest profits (e.g., 10% of managed assets), the fee amount would be about 2% of managed assets. The approximation error would be roughly `feeAmount / (M + P)`, or about 2% in this example.

While this approximation is common in ERC4626 vaults to avoid iterative calculations, it results in the protocol consistently receiving slightly less in fees than configured. Over time and with larger profits, this could accumulate to material amounts.

5 of 15 Findings — contracts/HakoStableVault.sol

Local Variable Shadowing

Low Risk

Detects local variables that shadow inherited functions from parent contracts, potentially causing unexpected behavior and confusion.

Shadowed variables:
- `name` parameter at line 59 in `initialize(string,string,address,address[])` shadows `ERC20Upgradeable.name()` and `IERC20Metadata.name()` functions
- `symbol` parameter at line 60 in `initialize(string,string,address,address[])` shadows `ERC20Upgradeable.symbol()` and `IERC20Metadata.symbol()` functions
- `decimals` local variable at line 818 in `_addAllowedDepositToken(address)` shadows `ERC20Upgradeable.decimals()` and `IERC20Metadata.decimals()` functions
- `allowance` local variable at line 495 in `depositIntoExternalVault(address,uint256)` shadows `ERC20Upgradeable.allowance(address,address)` and `IERC20.allowance(address,address)` functions

These shadowing instances can lead to confusion and make the code harder to maintain, as developers may accidentally reference the wrong function or variable.

6 of 15 Findings — contracts/HakoStableGateway.sol contracts/HakoStableVault.sol

Incorrect `nonReentrant` Modifier Placement

Low Risk

The `nonReentrant` modifier is not placed as the first modifier in several functions across both contracts. Best practice recommends placing the `nonReentrant` modifier first in the modifier list to protect against reentrancy attacks in other modifiers.

HakoStableGateway.sol: 6 functions with `nonReentrant` modifier not in first position

HakoStableVault.sol: 11 functions with `nonReentrant` modifier not in first position

Placing `nonReentrant` first ensures that the reentrancy guard is established before any other modifier logic executes, providing stronger protection against potential reentrancy vulnerabilities.

7 of 15 Findings contracts/HakoStableGateway.sol contracts/HakoStableVault.sol

Empty Code Block Detection

Low Risk

Both contracts contain empty function implementations that serve no purpose:

HakoStableGateway.sol:
- `_authorizeUpgrade(address)` function with empty body

HakoStableVault.sol:
- `_authorizeUpgrade(address)` function with empty body

These empty functions appear to be override implementations required by the upgrade mechanism but contain no logic. While they may be necessary for the contract structure, they should be reviewed to ensure they are intentional and not placeholders for missing functionality.

8 of 15 Findings contracts/HakoStableGateway.sol contracts/HakoStableVault.sol

Loop Contains `require` / `revert` Statements

Low Risk

Multiple functions in both contracts contain `require` or `revert` statements within loop structures. This pattern is problematic because a single invalid item in the loop can cause the entire transaction to fail, reverting all changes and preventing partial processing.

HakoStableGateway.sol: 2 loops with require/revert statements

HakoStableVault.sol: 2 loops with require/revert statements

Best practice recommends collecting failed items during loop iteration and returning them for post-processing rather than reverting the entire transaction. This allows the transaction to complete successfully while still identifying problematic items for the caller to handle separately.

9 of 15 Findings

contracts/HakoProxy.sol contracts/HakoStableGateway.sol contracts/HakoStableVault.sol
contracts/shared/VaultErrors.sol contracts/shared/VaultEvents.sol
contracts/shared/VaultMath.sol contracts/shared/VaultTypes.sol
contracts/stable/HakoStableGatewayStorage.sol
contracts/stable/HakoStableGatewayTypes.sol contracts/stable/HakoStableVaultRegistry.sol
contracts/stable/HakoStableVaultStorage.sol contracts/stable/HakoStableVaultTypes.sol

PUSH0 Opcode Compatibility Issue

• Low Risk

All contracts use Solidity version 0.8.30 or higher with the caret pragma (`pragma solidity ^0.8.30;`). Starting from Solc compiler version 0.8.20, the default target EVM version is Shanghai, which includes support for the PUSH0 opcode. However, not all blockchain networks support the PUSH0 opcode, particularly Layer 2 solutions and alternative EVM chains.

If these contracts are intended to be deployed on chains that do not support PUSH0 (such as Arbitrum, Optimism, or other L2 networks), the deployment will fail. It is essential to explicitly configure the EVM target version in the compiler settings to a version that does not include PUSH0 (such as Paris or earlier) when deploying to incompatible networks.

10 of 15 Findings

contracts/HakoProxy.sol contracts/HakoStableGateway.sol contracts/HakoStableVault.sol
contracts/shared/VaultErrors.sol contracts/shared/VaultEvents.sol
contracts/shared/VaultTypes.sol contracts/stable/HakoStableGatewayTypes.sol
contracts/stable/HakoStableVaultRegistry.sol contracts/stable/HakoStableVaultTypes.sol

Non-Specific Solidity Pragma Version

• Low Risk

All contracts use a wide pragma version range (`pragma solidity ^0.8.30;`) instead of a specific version. Using a caret (^) allows the compiler to use any version from 0.8.30 up to (but not including) 0.9.0, which can lead to inconsistent compilation behavior across different environments and potentially introduce unexpected compiler changes.

It is recommended to use a specific Solidity version (e.g., `pragma solidity 0.8.30;`) to ensure consistent and reproducible builds across all development and deployment environments.

11 of 15 Findings — contracts/shared/VaultMath.sol

Magic Numbers Instead Of Constants

Low Risk

The magic number `18` is used multiple times in calculations without being defined as a named constant. Specifically, the expression `10 ** (18 - decimals_)` appears in multiple locations within the contract.

Using magic numbers makes the code less readable and maintainable. Creating a constant state variable to represent this value would improve code clarity and make future modifications easier. The value `18` likely represents the standard EVM decimal precision (wei), and should be explicitly named to convey its purpose.

12 of 15 Findings — contracts/shared/VaultMath.sol

Unoptimized Numeric Literal Format

Low Risk

The numeric literal `10_000` is used in its full form instead of scientific notation in the expression `(taxableProfit * performanceFeeBps) / 10_000`. Using scientific notation (e.g., `1e4`) would be more concise and consistent with Solidity best practices for representing large numbers.

While the underscore separator improves readability, using exponential notation is the preferred format for powers of 10 in Solidity code.

13 of 15 Findings

contracts/HakoStableVault.sol

`recordRemoteDeposit*` allows privileged relayer to mint unbacked LP shares and inflate `totalManagedAssets` without any onchain proof of remote funds

Info

`HakoStableVault.recordRemoteDeposit(...)` and `HakoStableVault.recordRemoteDepositNonEvm(...)` mint LP shares and increase `totalManagedAssets` based purely on the `amountNormalized` argument supplied by `RELAYER_ROLE`. Aside from replay protection on `depositId`, there is no onchain verification that any remote-chain gateway received assets, that a bridge occurred, or that the normalized amount corresponds to an observed event.

Code path (share mint + accounting increase happens unconditionally after the replay check):

```
function recordRemoteDeposit(bytes32 depositId, address receiver, uint256 amountNormalized)
    external
    onlyRole(RELAYER_ROLE)
    nonReentrant
    whenNotPaused
    returns (uint256 sharesMinted)
{
    if (receiver == address(0)) revert ZeroAddress();

    HakoStableVaultStorage.Layout storage layout_ = HakoStableVaultStorage.layout();
    if (layout_.processedDeposits[depositId]) revert DepositAlreadyProcessed(depositId);
    layout_.processedDeposits[depositId] = true;

    sharesMinted = _mintSharesForDeposit(receiver, amountNormalized);
    emit DepositRecorded(depositId, receiver, amountNormalized, sharesMinted, true);
}

function _mintSharesForDeposit(address receiver, uint256 amountNormalized) internal returns
(uint256 sharesMinted) {
    ...
    sharesMinted = VaultMath.previewMintShares(amountNormalized, supply, managed);
    ...
    layout_.totalManagedAssets = managed + amountNormalized;
    _mint(receiver, sharesMinted);
}
```

Impact: any compromise/misbehavior of the `RELAYER_ROLE` can (i) dilute all existing LP holders by minting arbitrary shares to an attacker-controlled receiver and (ii) corrupt the canonical accounting (`totalManagedAssets`), which then propagates into share-price calculations and withdrawal share-lock calculations. This risk exists even if custody assets on gateways are unchanged, because the vault's accounting is authoritative for LP share value.

14 of 15 Findings

contracts/HakoStableVault.sol

Withdrawal finalization burns user shares and reduces `totalManagedAssets` without any onchain linkage to payout execution, enabling irrevocable loss if the finalizer misbehaves

Info

`HakoStableVault.completeWithdrawal(...)` finalizes a pending withdrawal by (a) decrementing `lockedShares`, (b) decrementing `totalManagedAssets`, and (c) burning the owner's shares, but it does not perform (or verify) any payout transfer on the home chain or any proven cross-chain settlement.

Relevant logic:

```
function completeWithdrawal(uint256 requestId) external onlyRole(WITHDRAW_FINALIZER_ROLE)
nonReentrant {
    HakoStableVaultStorage.Layout storage layout_ = HakoStableVaultStorage.layout();
    WithdrawalRequest storage req = layout_.withdrawalRequests[requestId];
    if (req.status != WithdrawalStatus.Pending) revert WithdrawalNotPending(requestId);

    req.status = WithdrawalStatus.Completed;

    layout_.lockedShares[req.owner] -= req.sharesLocked;
    if (layout_.totalManagedAssets < req.amountNormalized) revert ManagedAssetsUnderflow();
    layout_.totalManagedAssets -= req.amountNormalized;

    _burn(req.owner, req.sharesLocked);

    emit WithdrawalCompleted(requestId, req.owner, req.sharesLocked, req.amountNormalized);
}
```

Impact: if `WITHDRAW_FINALIZER_ROLE` is compromised or operationally incorrect, it can irreversibly destroy a user's LP shares (and reduce managed-assets accounting) without any onchain enforcement that the user is actually paid out on the destination chain. Because this burn/accounting action is the canonical source of truth, the user cannot recover their locked shares once completed, even if no payout occurs.

15 of 15 Findings contracts/HakoStableVault.sol contracts/HakoStableGateway.sol

Unlimited ERC20 allowances granted to external ERC-4626 vaults allow the vault contract to pull arbitrary underlying (drain) if the external vault becomes malicious or is upgraded

Info

Both `HakoStableVault.depositIntoExternalVault(...)` and `HakoStableGateway.depositIntoExternalVault(...)` use `forceApprove(vault, type(uint256).max)` when the current allowance is insufficient. This permanently grants the external vault contract an effectively unlimited allowance over the underlying token balance held by the Hako contract.

Example (same pattern in both contracts):

```
IERC20 underlying = IERC20(asset);
uint256 allowance = underlying.allowance(address(this), vault);
if (allowance < assets) {
    underlying.forceApprove(vault, type(uint256).max);
}

sharesMinted = IERC4626(vault).deposit(assets, address(this));
```

Although external vaults are allowlisted via `setExternalVaultAllowed`, the allowance is a standing approval to the *vault contract address*, not to a specific immutable strategy implementation. If the ERC-4626 vault is upgradeable, compromised, or maliciously configured, it can call `transferFrom` on the underlying at any time (not only during `deposit`), draining all underlying tokens held by `HakoStableVault` / `HakoStableGateway`, bypassing the receiver restrictions used in withdraw/redeem flows.

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.