Pashov Audit Group

Nucleus Security Review

February 3rd 2026 - February 5th 2026

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Nucleus

Nucleus PR 237 introduces a withdraw queue with ERC-721 receipts: FIFO processing, visible order statuses, signature/permit deposits, pre-fill/refund paths, multi-asset teller integration, and configurable fee routing.

5. Executive Summary

A time-boxed security review of the **Ion-Protocol/nucleus-boring-vault** repository was done by Pashov Audit Group, during which **Hals**, **BenRai**, **Rayaa**, **BengalCatBalu** engaged to review **Nucleus**. A total of **8** issues were uncovered.

Protocol Summary	
Project Name	Nucleus
Protocol Type	Vault Management
Timeline	February 3rd 2026 - February 5th 2026

Review commit hash:

- a8e05904b5c31da71b6047c6a905cd25640cb9b7 (Ion-Protocol/nucleus-boring-vault)

Fixes review commit hash:

- 1b8660ecbfe317f226ff6e8730f79a07caa950d8 (Ion-Protocol/nucleus-boring-vault)

Scope

`WithdrawQueue.sol` `TellerWithMultiAssetSupport.sol` `CrossChainTellerBase.sol` `AccountantWithRateProviders.sol`

6. Findings

Findings count

Severity	Amount
Low	8
Total findings	**8**

Summary of findings

ID	Title	Severity	Status
[L-01]	Manage ERC20 does not validate zero amounts	Low	**Resolved**
[L-02]	Division before multiplication in claimFees causes fee loss	Low	Acknowledged
[L-03]	No revert when feesOwedInFeeAsset rounds down to zero	Low	**Resolved**
[L-04]	Accountant pause creates inconsistent behavior	Low	Acknowledged
[L-05]	Performance fee changes apply immediately without grace period	Low	Acknowledged
[L-06]	Gains are taxed twice by management and performance fees	Low	Acknowledged
[L-07]	Performance fee not charged for users depositing below highest exchange rate	Low	Acknowledged
[L-08]	Sequential nonce usage causes dependency	Low	Acknowledged

Low findings

[L-01] Manage ERC20 does not validate zero amounts

Description

`manageERC20()` checks for zero addresses but not for zero amounts. Calling it with `amount == 0` performs a no-op transfer and wastes gas. More importantly, it can mask configuration errors where the caller intended to specify a non-zero amount. Adding a zero-amount check improves consistency with the zero-address checks and prevents accidental no-ops.

Recommendation

Add a zero-amount check at the start of the function:

```
if (amount == 0) revert ZeroAmount();
```

[L-02] Division before multiplication in claimFees causes fee loss

Description

In `AccountantWithRateProviders.claimFees()`, when the fee asset is not pegged to base, `feesOwedInBase` is first scaled via `changeDecimals()` (which may divide) and then multiplied by the rate. This division-before-multiplication pattern can truncate small amounts and cause fee loss. Prefer using `feesOwedInBase` directly in the rate conversion without scaling down first, or ensure multiplication occurs before division to preserve precision.

[L-03] No revert when feesOwedInFeeAsset rounds down to zero

Description

In `AccountantWithRateProviders.claimFees()`, after computing `feesOwedInFeeAsset`, the contract does not check if it is zero. Due to rounding down in the conversion (especially with division-before-multiplication), the computed amount can be zero even when `feesOwedInBase` is non-zero. The contract would then zero out `feesOwedInBase` and transfer 0 tokens, permanently losing the accrued fees. Add a check to revert if `feesOwedInFeeAsset == 0` when `state.feesOwedInBase > 0`.

[L-04] Accountant pause creates inconsistent behavior

Description

When the Accountant is paused, `getRateInQuoteSafe()` reverts:

```
// AccountantWithRateProviders.sol:485-487
function getRateInQuoteSafe(ERC20 quote) external view returns (uint256 rateInQuote) {
    if (accountantState.isPaused) revert AccountantWithRateProviders__Paused();
    rateInQuote = getRateInQuote(quote);
}
```

This function is called twice during order processing:

1. **Outside try-catch** (WithdrawQueue.sol:476) — for pre-calculating `expectedAssetsOut`
2. **Inside try-catch** via `bulkWithdraw` (TellerWithMultiAssetSupport.sol:337) — for actual withdrawal calculation

```
// WithdrawQueue.sol:475-484 — first call OUTSIDE try-catch
uint256 expectedAssetsOut = tellerWithMultiAssetSupport.accountant()
    .getRateInQuoteSafe(ERC20(address(order.wantAsset)))  // ← reverts here if paused
    .mulDivDown((order.amountOffer - feeAmount), 10 ** vault.decimals());
// ...
try tellerWithMultiAssetSupport.bulkWithdraw(...) {  // ← second call INSIDE try-catch
```

If the Accountant becomes paused between `processOrders` calls or mid-batch, the behavior is undefined:

- Now it reverts on the first call (L476); the **entire batch fails** — no orders processed.
- In another way, it may revert inside `bulkWithdraw`, and then the order is **auto-refunded**.

This inconsistency means there is no clear answer to "what protocol wants to happen when the Accountant is paused during processing?" Unlike the Teller pause, which has an explicit `TellerIsPaused` check at L462 that blocks all DEFAULT orders cleanly.

Consider adding an explicit `accountant.isPaused()` check alongside the Teller pause check to create consistent behavior if you want to revert all orders processing. The current implicit handling via revert location is fragile and unclear. Instead, if you want to refund orders in a paused accountant state, call `getRateInQuote` instead of `getRateInQuoteSafe` before the try-catch block.

[L-05] Performance fee changes apply immediately without grace period

Description

`updatePerformanceFee()` and `updateManagementFee()` in `AccountantWithRateProviders` apply the new fee values immediately. Users who deposited under one fee regime are subject to the new fees on their next interaction (e.g. withdrawal or fee accrual) without prior notice or opportunity to exit.

This breaks the expectation that the fee terms at deposit time govern the user's position. A malicious or careless admin could raise the performance fee to 100% and capture all gains for existing depositors who had no way to consent to the change. Even with trusted admins, users may not monitor for fee updates and could be surprised by higher fees on withdrawal.

The same applies to `updateManagementFee()`. Both functions are `requiresAuth` and can be called at any time with no time lock or grace period.

Recommendation

Introduce a grace period after fee updates during which users can withdraw without the new fee. For example:

1. Store `pendingPerformanceFee` and `feeChangeTimestamp` when `updatePerformanceFee()` is called.
2. Only apply the new fee after `feeChangeTimestamp + GRACE_PERIOD` (e.g. 7 days).
3. During the grace period, continue using the previous fee for fee calculations.
4. Emit an event with the new fee and effective timestamp so users and integrators can react.

Alternatively, use a timelock pattern: require a two-step update (propose → execute after delay) so users have time to exit before the new fee takes effect.

[L-06] Gains are taxed twice by management and performance fees

Severity

Impact: Medium

Likelihood: Medium

Description

The fee model in `updateExchangeRate()` charges both management fees and performance fees on overlapping asset bases. Management fees are computed on `minimumAssets` (the minimum of current and new exchange rates applied to share supply), while performance fees

are computed on `changeInAssets` (the increase in asset value above `highestExchangeRate`). When the exchange rate rises, the same underlying gains can be subject to both fees.

Management fees are intended to be charged on assets under management (principal plus gains), and performance fees on gains only. The current implementation does not separate principal from gains. As a result, gains are effectively taxed twice: once via the management fee (which includes gains in its base) and again via the performance fee. This breaks the intended fee semantics and leads to overcharging users relative to a correct split between management and performance fees.

Recommendation

Track the total principal (deposited value) across all users. When computing fees:

- If total asset value ≤ principal: charge management fee on current asset value only.
- If total asset value > principal: charge management fee on principal value and performance fee on the gain above principal.

This requires maintaining a principal accounting mechanism (e.g., cumulative deposits minus withdrawals in base terms) and using it when calculating both fee components.

[L-07] Performance fee not charged for users depositing below highest exchange rate

Performance fees in `updateExchangeRate()` are only accrued when `newExchangeRate > state.highestExchangeRate`. The fee is computed on `changeInAssets`, which is the increase from `highestExchangeRate` to `newExchangeRate`. Users who deposit when the exchange rate is below the current `highestExchangeRate` receive shares at a lower price but are never charged a performance fee on their gains until the rate exceeds the previous high again.

Example: `highestExchangeRate = 1.5`, user deposits at `exchangeRate = 1` and receives 1000 shares for 1000 USD. The rate rises to 1.5. The user withdraws with a profit of 500 USD but pays no performance fee on this gain because the global `highestExchangeRate` was already 1.5 when they deposited. The protocol loses performance fee revenue.

Recommendation

Track the average "buy" price (or cost basis) of each user's shares. Charge the performance fee at withdrawal time based on the gain above that cost basis, rather than using a global `highestExchangeRate`. This ensures all gains are fairly taxed regardless of when the user deposited.

[L-08] Sequential nonce usage causes dependency

The submit-order signature in `_verifyDepositor()` uses a strictly sequential nonce:

```
bytes32 hash = keccak256(
    abi.encode(
        params.amountOffer,
        ...
        nonces[params.intendedDepositor]++,  // ← increments sequentially: 0, 1, 2, ...
        ...
    )
);
```

Unlike `OneToOneQueue`, which uses arbitrary user-chosen nonces with a `usedSignatureHashes` mapping:

```
// OneToOneQueue approach — flexible ordering
bytes32 hash = keccak256(abi.encode(..., params.nonce, ...));
if (usedSignatureHashes[hash]) revert SignatureAlreadyUsed();
usedSignatureHashes[hash] = true;
```

`WithdrawQueue` requires signatures to be submitted in the exact order they were created. This creates practical issues for meta-transaction workflows:

1. **Expired signature blocks subsequent ones**: If a user creates signature A (nonce 0, deadline 3 days) and signature B (nonce 1, deadline 1 day), signature B cannot be submitted until A is submitted first. If A expires or is no longer desired, all subsequent signatures (B, C, ...) must be recreated with new nonces.
2. **Failed submission invalidates the chain**: If a relayer attempts to submit signature A but the transaction reverts (e.g., insufficient shares, asset no longer supported), the nonce is not consumed. However, if A is submitted successfully, nonce 0 is consumed and B becomes available. If A was intended to be optional, there is no way to skip it and submit B directly.
3. **No out-of-order execution**: Two independent users creating signatures for the same depositor (e.g., through a shared multisig or relayer) must coordinate nonce ordering off-chain.

The predecessor `OneToOneQueue` solved this with a more flexible approach: the nonce is an arbitrary user-chosen value, and replay protection is handled by a `usedSignatureHashes` mapping that records consumed hashes regardless of ordering.

Recommendations

Adopt the `OneToOneQueue` pattern where the nonce is user-chosen and replay protection uses a `usedSignatureHashes` mapping instead of sequential enforcement. This allows signatures to be submitted in any order and eliminates the cascading invalidation problem.