Pashov Audit Group

Nucleus Security Review

November 22nd 2025 - November 26th 2025

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Risk Classification 3
4. About Nucleus 4
5. Executive Summary 4
6. Findings 5

Medium findings 6

[M-01] Signature-based orders lack fee config 6

Low findings 8

[L-01] `getOrderStatus(0)` incorrectly reports a completed order that never existed 8
[L-02] `DeprecationBegun` event is never emitted 8
[L-03] `CommunityCodeDepositor` is not compliant with zero-reset approvals 8
[L-04] Double-spend of `wantAsset` possible when non-standard ERC20 returns false 9
[L-05] Interface implemented without `override` 9
[L-06] Cross-chain hash collisions possible if backend relies on `depositHash` uniqueness 10
[L-07] Inconsistent error handling in `setAuthority` 11
[L-08] Shared deadline variable for different signatures 11
[L-09] Blacklisted `refundReceiver` can freeze order processing when `wantAsset` is deprecated 12
[L-10] Refund receiver unupdated on NFT transfer risks refund loss 13
[L-11] Asymmetric auth consumption in `_verifyDepositor` causes inconsistency 14
[L-12] Nonce reuse allows unauthorized orders causing asset loss 15

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 CommunityCodeDepositor and OneToOneQueue introduce community code–based deposit flows with permit support, shared vault architecture updates, and a new FIFO one-to-one queue using ERC721 receipts to track deposits and withdrawals across native and ERC20 assets. The update also adds enhanced access control with pausing and staged deprecation, configurable fee handling, recovery for failed transfers, and broader multi-asset support.

5. Executive Summary

A time-boxed security review of the **Ion-Protocol/nucleus-boring-vault** and **Ion-Protocol/nucleus-boring-vault** repositories was done by Pashov Audit Group, during which **0xAlix2**, **Hals**, **BenRai**, **0xbepresent**, **BengalCatBalu** engaged to review **Nucleus.** A total of **13** issues were uncovered.

Protocol Summary	
Project Name	Nucleus
Protocol Type	Vault Management
Timeline	November 22nd 2025 - November 26th 2025

Review commit hashes:

- 7032978ef1fc44271145bb593be45be1c7a0b6c5 (Ion-Protocol/nucleus-boring-vault)
- c3214ca65cd9bc38bc0d23a1dbb965474b64915f (Ion-Protocol/nucleus-boring-vault)

Fixes review commit hash:

- ef5cf37acc8d073d6e25953e68eec8b663808ae7 (Ion-Protocol/nucleus-boring-vault)

Scope

`CommunityCodeDepositor.sol` `DistributorCodeDepositor.sol` `OneToOneQueue.sol` `QueueAccessAuthority.sol` `SimpleFeeModule.sol` `AccessAuthority.sol` `Pausable.sol` `VerboseAuth.sol` `IAccessAuthorityHook.sol` `IFeeModule.sol`

6. Findings

Findings count

Severity	Amount
Medium	1
Low	12
Total findings	**13**

Summary of findings

ID	Title	Severity	Status
[M-01]	Signature-based orders lack fee config	Medium	**Resolved**
[L-01]	`getOrderStatus(0)` incorrectly reports a completed order that never existed	Low	**Resolved**
[L-02]	`DeprecationBegun` event is never emitted	Low	**Resolved**
[L-03]	`CommunityCodeDepositor` is not compliant with zero-reset approvals	Low	**Resolved**
[L-04]	Double-spend of `wantAsset` possible when non-standard ERC20 returns false	Low	Acknowledged
[L-05]	Interface implemented without `override`	Low	**Resolved**
[L-06]	Cross-chain hash collisions possible if backend relies on `depositHash` uniqueness	Low	**Resolved**
[L-07]	Inconsistent error handling in `setAuthority`	Low	**Resolved**
[L-08]	Shared deadline variable for different signatures	Low	Acknowledged
[L-09]	Blacklisted `refundReceiver` can freeze order processing when `wantAsset` is deprecated	Low	**Resolved**
[L-10]	Refund receiver unupdated on NFT transfer risks refund loss	Low	Acknowledged
[L-11]	Asymmetric auth consumption in `_verifyDepositor` causes inconsistency	Low	Acknowledged
[L-12]	Nonce reuse allows unauthorized orders causing asset loss	Low	Acknowledged

Medium findings

[M-01] Signature-based orders lack fee config

Severity

Impact: High

Likelihood: Low

Description

When orders are submitted via signature, `_verifyDepositor` validates an off-chain signature over the following fields:

```
bytes32 hash = keccak256(
    abi.encode(
        params.amountOffer,
        params.offerAsset,
        params.wantAsset,
        params.receiver,
        params.refundReceiver,
        params.signatureParams.deadline,
        params.signatureParams.approvalMethod,
        params.signatureParams.nonce,
        block.chainid,
        address(this)
    )
);
```

However, the signature does **not** commit to any fee-related parameter or identifier, such as:

- The current `feeModule` address.
- Any fee rate.

Later, in `submitOrder`, the contract computes:

```
uint256 feeAmount = feeModule.calculateOfferFees(params.amountOffer, params.offerAsset,
params.wantAsset, params.receiver);
uint256 newAmountForReceiver = params.amountOffer - feeAmount;
```

Because `feeModule` is mutable via `setFeeModule`, and fee logic inside the module can also be upgradeable/configurable, the following scenario is possible:

1. A user signs an order off-chain while fees are low (e.g. 1%) and gives the signature to a trusted relayer/front-end.
2. Before the signature is submitted on-chain, the owner updates `feeModule` or its parameters to increase fees (e.g. 5%).
3. The relayer submits the *same* signature.

4. The signature is still valid (hash unchanged), but the on-chain execution now charges the higher fee; the user has no on-chain protection ensuring that their signed intent was “1% fee, not 5%”.

This results in users submitting orders via signatures can be charged significantly higher fees than intended if the `feeModule` or its fee schedule changes between signature creation and execution.

Recommendations

Consider expposing some kind of an identifier in the fee module (includes current fee percentage) that could be included later in the signature's hash:

```
bytes32 hash = keccak256(
    abi.encode(
        params.amountOffer,
        params.offerAsset,
        params.wantAsset,
        params.receiver,
        params.refundReceiver,
        params.signatureParams.deadline,
        params.signatureParams.approvalMethod,
        params.signatureParams.nonce,
+       feeModule.identifier(),
        block.chainid,
        address(this)
    )
);
```

Low findings

[L-01] `getOrderStatus(0)` incorrectly reports a completed order that never existed

The queue is strictly **1-indexed**, but `getOrderStatus(0)` returns `OrderStatus.COMPLETE` because `orderIndex > lastProcessedOrder` is false when index is `0`. This makes order `0` appear as a valid, fully processed order even though no such order is ever created or stored in the queue.

Recommendations:

Consider adding an explicit check that treats `orderIndex == 0` as invalid or `NOT_FOUND`.

[L-02] `DeprecationBegun` event is never emitted

`AccessAuthority` declares a `DeprecationBegun(uint8 step)` event but never emits it. Deprecation state transitions are only signalled via `DeprecationContinued` and `DeprecationFinished` in `continueDeprecation()`, so there is no dedicated on-chain signal for the initial transition from "not deprecated" to "deprecated".

This makes the `DeprecationBegun` event effectively dead code.

Recommendations:

Consider emitting the `DeprecationBegun` event in `continueDeprecation()` when `deprecationStep == 0`.

[L-03] `CommunityCodeDepositor` is not compliant with zero-reset approvals

The documentation states:

> "We want to be sure we can handle deposits for any kind of ERC20 the world throws at us."

However, the `_deposit` logic unconditionally performs `safeApprove(spender, 0)` before setting a new allowance. ERC20 tokens that revert on zero-value approvals will always revert during this step, preventing deposits. This contradicts the stated goal of broad ERC20 compatibility.

Recommendations:

Consider wrapping 0-approval with a try/catch.

[L-04] Double-spend of `wantAsset` possible when non-standard ERC20 returns false

In the `OneToOneQueue.processOrders()` function, it is **acknowledged** that some ERC20 tokens (such as gold-backed tokens) may return `false` while still performing a successful transfer. However, the processing logic still treats this as a failed transfer and sends the same `amountWant` again to the `recoveryAddress`. This results in the `receiver` getting the tokens once from the actual transfer and once from the recovery flow, allowing double-spending of the `wantAsset`.

```
 function processOrders(uint256 ordersToProcess) public requiresAuthVerbose {
        //...
        for (uint256 i; i < ordersToProcess; ++i) {
           //...

@514>       // From SafeERC20 library to perform a safeERC20 transfer and return a bool on
success. Implemented here
            // since it's private and we cannot call it directly It's worth noting that there
are some tokens like
            // Tether Gold that return false while succeeding.
            // This situation would result success in being false even though the transfer did
not revert.
@518>       bool success = _callOptionalReturnBool(order.wantAsset, receiver, order.amountWant);

@525>       if (!success) {
                // Set the type for the storage and memory as we will emit the memory order
                order.orderType = OrderType.FAILED_TRANSFER;
                queue[orderIndex].orderType = OrderType.FAILED_TRANSFER;
@529>           order.wantAsset.safeTransfer(recoveryAddress, order.amountWant);
                emit OrderFailedTransfer(orderIndex, recoveryAddress, receiver, order);
            }

            //...
        }

        //...
    }
```

Recommendation: Use a transfer mechanism that verifies actual balance changes.

[L-05] Interface implemented without `override`

`SimpleFeeModule` implements `IFeeModule`, but its `calculateOfferFees()` function is missing the required `override` keyword, causing the contract to fail compilation.

```
// @interfaces/IFeeModule.sol
interface IFeeModule {
    function calculateOfferFees(
        uint256 amount,
        IERC20 offerAsset,
        IERC20 wantAsset,
        address receiver
    )
        external
        view
        returns (uint256 feeAmount);
}
```

```
// @one-to-one-queue/SimpleFeeModule.sol
contract SimpleFeeModule is IFeeModule {
    //...
    function calculateOfferFees(
        uint256 amount,
        IERC20 offerAsset,
        IERC20 wantAsset,
        address receiver
    )
        external
        view
        returns (uint256 feeAmount)
    {
        //...
    }

}
```

Recommendation: Add the `override` keyword to the `calculateOfferFees()` function.

[L-06] Cross-chain hash collisions possible if backend relies on `depositHash` uniqueness

The `CommunityCodeDepositor._deposit()` generates a `depositHash` using only the contract address and a local incrementing nonce. If the `CommunityCodeDepositor` is deployed to the same address on multiple chains, deposits made on different chains may produce identical event parameters, including identical `depositHash` values.

```
    function _deposit(
        ERC20 depositAsset,
        uint256 depositAmount,
        uint256 minimumMint,
        address to,
        bytes calldata communityCode
    )
        internal
        returns (uint256 shares)
    {
        //...
        unchecked {
@180>       depositHash = keccak256(abi.encodePacked(address(this), ++depositNonce));
        }

        //...
@192>   emit DepositWithCommunityCode(
            msg.sender, depositAsset, depositAmount, minimumMint, to, depositHash, communityCode
        );
    }
```

This becomes an issue if the backend system relies on `depositHash` as a globally unique identifier for referral tracking and community reward attribution, in such a case, identical operations on different chains may produce identical deposit hashes causing incorrect attribution of referral rewards.

If the backend uses `(chainId, txHash, logIndex)` to identify events, then this is not an issue, as the risk exists only when `depositHash` is treated as a standalone unique key.

Recommendation: Ensure that each emitted deposit event is uniquely identifiable across chains by including `chain.id` when generating the `depositHash` .

[L-07] Inconsistent error handling in `setAuthority`

The `VerboseAuth` contract contains an inconsistency in error handling between the `setAuthority` function and the `isAuthorizedVerbose` function when the authority is set to `address(0)` .

In `setAuthority` , when a non-owner calls the function and the current authority is `address(0)` , the code attempts to call `authority.canCallVerbose()` on a zero address:

```
File: VerboseAuth.sol
    function setAuthority(Authority newAuthority) public virtual {
...
        if (msg.sender != owner) {
@>          (bool canCall,) = authority.canCallVerbose(msg.sender, address(this), msg.data);
            require(canCall);
        }
```

This results in a low-level revert when calling a function on `address(0)` .

However, in `isAuthorizedVerbose` , the code properly handles the `address(0)` case:

```
File: VerboseAuth.sol
        Authority auth = Authority(address(authority));

        if (address(auth) == address(0)) return (false, "- No Authority Set: Owner Only ");
```

Fix the inconsistency by adding the same `address(0)` check in `setAuthority` that exists in `isAuthorizedVerbose` . This ensures consistent error handling across all authorization functions in the VerboseAuth contract.

[L-08] Shared deadline variable for different signatures

The `submitOrder` function can be called using a signature. Additionally, the allowance for the offer asset can be granted using an ERC-2612 permit. This means two different signatures are required: one for the function call and another for the permit. However, the protocol uses the same deadline for both signatures.

Signature verification is performed in the `_verifyDepositor` function — it first checks that the signature used for calling the function has not expired its deadline.

```
 if (block.timestamp > params.signatureParams.deadline) {
                revert SignatureExpired(params.signatureParams.deadline, block.timestamp);
}
```

After that, the same deadline is passed into the permit function call.

```
try IERC20Permit(address(params.offerAsset))
        .permit(
            depositor,
            address(this),
            params.amountOffer,
            params.signatureParams.deadline, // @audit here
            params.signatureParams.approvalV,
            params.signatureParams.approvalR,
            params.signatureParams.approvalS
        ) { }
catch {
        if (params.offerAsset.allowance(depositor, address(this)) < params.amountOffer) {
            revert PermitFailedAndAllowanceTooLow();
        }
}
```

This approach requires the user to create two new signatures instead of one if only a single deadline expires, and overall imposes additional constraints on user actions. As a result, if a deadline expires due to, for example, the `OneToOneQueue` being paused, the user will be forced to generate an additional signature for the permit. Conversely, if the deadline expires due to issues on the token contract's side, the user will need to create another signature for the `OneToOneQueue`.

Since this issue only introduces inconvenience and limitations for the user and does not lead to more serious consequences, it is classified as low severity.

[L-09] Blacklisted `refundReceiver` can freeze order processing when `wantAsset` is deprecated

In `OneToOneQueue._forceRefund()`, refunds are sent directly to the order's fixed `refundReceiver` address without any fallback. If the `refundReceiver` is blacklisted or blocked by the `offerAsset`, the refund transfer reverts and **cannot succeed**, making the order permanently unrefundable.

```
    function _forceRefund(uint256 orderIndex) internal {
        Order memory order = _getOrderEnsureDefault(orderIndex);

        // Mark as refunded
        queue[orderIndex].orderType = OrderType.REFUND;

        _checkBalanceQueue(order.offerAsset, order.amountOffer, orderIndex);
        _burn(orderIndex);
@663>   order.offerAsset.safeTransfer(order.refundReceiver, order.amountOffer);
```

```
        emit OrderRefunded(orderIndex, queue[orderIndex]);
    }
```

This becomes more severe when the `wantAsset` has been removed from the `supportedWantAssets` list and the protocol intends to refund all remaining orders for that asset.

Because the refund flow cannot bypass a blacklisted `refundReceiver`, the only remaining way to clear such orders is to process them normally through `processOrders()`. However, this requires sufficient liquidity of the order's `wantAsset`. If the contract no longer supports swapping that asset and (as a result) no longer holds enough of it, the order **cannot be processed either**.

In this scenario, the refund path is blocked (`refundReceiver` blacklisted), and the processing path is blocked (insufficient `wantAsset` liquidity), and since `processOrders()` increments `lastProcessedOrder` sequentially, the stuck order (and failed-to-be-refunded orders) prevents all subsequent orders from being processed, creating a queue-wide DoS.

Recommendations

Add a refund fallback mechanism similar to the `FAILED_TRANSFER` logic used in `processOrders()`: if a refund transfer fails, redirect the refund to a recovery address and mark the order accordingly.

[L-10] Refund receiver unupdated on NFT transfer risks refund loss

`OneToOneQueue` order receipts are transferable ERC721 tokens and can be freely traded on secondary markets. However, each order stores a fixed `refundReceiver` set by the original submitter. If the NFT is later sold, the new buyer acquires the economic rights to the order, but does not become the refund recipient and can't update this refund recipient either.

When the order is refunded, the assets are still sent to the original `refundReceiver`, not the current NFT owner:

```
    function _forceRefund(uint256 orderIndex) internal {
        Order memory order = _getOrderEnsureDefault(orderIndex);

        // Mark as refunded
        queue[orderIndex].orderType = OrderType.REFUND;

        _checkBalanceQueue(order.offerAsset, order.amountOffer, orderIndex);
        _burn(orderIndex);
@663>   order.offerAsset.safeTransfer(order.refundReceiver, order.amountOffer);

        emit OrderRefunded(orderIndex, queue[orderIndex]);
    }
```

Recommendations

Bind refunds to the current NFT owner or allow the current NFT owner to update the `refundReceiver` address.

[L-11] Asymmetric auth consumption in `_verifyDepositor` causes inconsistency

The `OneToOneQueue` contract's `_verifyDepositor` function implements dual authorization mechanisms: ECDSA signature verification for order submission and EIP-2612 PERMIT for token approval. However, these mechanisms are consumed asymmetrically, creating an inconsistent authorization state.

When a user submits an order with both authorization methods:

```
File: OneToOneQueue.sol
546:     function _verifyDepositor(SubmitOrderParams calldata params) internal returns (address
depositor) {
547:         if (params.signatureParams.submitWithSignature) {
...
551:             bytes32 hash = keccak256(
...
565:             if (usedSignatureHashes[hash]) revert SignatureHashAlreadyUsed(hash);
566:@>           usedSignatureHashes[hash] = true;
...
581:         if (params.signatureParams.approvalMethod == ApprovalMethod.EIP2612_PERMIT) {
582:@>           try IERC20Permit(address(params.offerAsset))
583:                 .permit(
...
```

The asymmetric consumption problem

1. **ECDSA Signature**: Gets marked as used in `usedSignatureHashes` (consumed/prevented from reuse line 566).
2. **PERMIT Authorization**: Sets token allowance (line 582) but doesn't consume it after the transfer.

Consider an aggregator with dual authorizations:

1. **User creates ECDSA signature** for order: `amountOffer = 10`, `nonce = 1`.
2. **User creates PERMIT signature** approving contract to spend: `amountOffer = 10`.
3. **Aggregator submits order**:
4. ECDSA signature verified and marked as used ✓
5. PERMIT executed, setting 10 token allowance ✓
6. Contract transfers 10 tokens from user ✓

`solidity File: OneToOneQueue.sol 460: params.offerAsset.safeTransferFrom(depositor, offerAssetRecipient, newAmountForReceiver);`

User has transferred 10 tokens, but still has 10 tokens worth of PERMIT allowance available for future use.

Impact:

- **Inconsistent Authorization State**: One authorization mechanism is consumed while the other remains active.
- **User Confusion**: Users expecting dual authorization to require both mechanisms may be surprised when only one is consumed.
- **Potential Asset Exposure**: Leftover PERMIT allowances could be exploited in subsequent transactions.

Recommendations

If dual authorization is intended, modify the logic to require both mechanisms and consume both. This fix ensures consistent authorization behavior and prevents the asymmetric consumption issue that could lead to unexpected protocol behavior.

[L-12] Nonce reuse allows unauthorized orders causing asset loss

The `OneToOneQueue` contract implements a signature-based system intended to enable gasless submissions where users can delegate order execution to third parties (aggregators). However, the current implementation only tracks complete signature hashes rather than enforcing `per-user nonce uniqueness`. This allows an approved aggregator to create multiple valid signatures using the same `nonce` but with different order parameters (such as `amountOffer`).

```
File: OneToOneQueue.sol
            }
            bytes32 hash = keccak256(
                abi.encode(
                    params.amountOffer,
                    params.offerAsset,
                    params.wantAsset,
                    params.receiver,
                    params.refundReceiver,
                    params.signatureParams.deadline,
                    params.signatureParams.approvalMethod,
@>                  params.signatureParams.nonce,
                    block.chainid,
                    address(this)
                )
            );
            if (usedSignatureHashes[hash]) revert SignatureHashAlreadyUsed(hash);
@>          usedSignatureHashes[hash] = true;
```

Consider the following scenario:

1. **User Signs Initial Message**: User approves aggregator and signs order with `nonce = 1`, `amountOffer = 200` → `hash1 = keccak256(abi.encode(200, ..., 1, ...))`

2. **User Attempts Cancellation**: User wants to cancel the first message and signs new order with same `nonce = 1` but `amountOffer = 100` → `hash2 = keccak256(abi.encode(100, ..., 1, ...))`
3. **Both Signatures Valid**: Both signatures are cryptographically valid since they use the same nonce from the same user. The system cannot prevent either from being executed.
4. **Race Condition Execution**: Aggregator submits both orders simultaneously:
5. Order 1: `nonce = 1`, `amountOffer = 200` (succeeds)
6. Order 2: `nonce = 1`, `amountOffer = 100` (also succeeds because `hash2 ≠ hash1`)
7. **No Cancellation Possible**: User cannot revoke the nonce permission once granted. Both orders execute, resulting in 300 tokens being traded instead of the intended single trade.

Impact:

- **No Cancellation Mechanism**: Users cannot cancel or invalidate signed messages once approved
- **Aggregator Exploitation**: Malicious aggregators can trick (or by chance) users into signing multiple messages with the same nonce

Recommendations

Implement Per-User Nonce Tracking. Replace or supplement the signature hash tracking with per-user nonce management:

```
mapping(address => mapping(uint256 => bool)) public usedNonces;

function _verifyDepositor(SubmitOrderParams calldata params) internal returns (address
depositor) {
    if (params.signatureParams.submitWithSignature) {
        // ... existing deadline check ...

        // Check if nonce already used by this depositor
        if (usedNonces[params.intendedDepositor][params.signatureParams.nonce]) {
            revert NonceAlreadyUsed(params.intendedDepositor, params.signatureParams.nonce);
        }

        // ... existing hash creation and signature verification ...

        // Mark nonce as used AFTER successful verification
        usedNonces[params.intendedDepositor][params.signatureParams.nonce] = true;

        // Keep existing usedSignatureHashes for additional protection
        usedSignatureHashes[hash] = true;
    }
}
```

Additionally. Add nonce cancellation functionality, implement functions allowing users to invalidate future nonces:

```
function cancelNonce(uint256 nonce) external {
    usedNonces[msg.sender][nonce] = true;
    emit NonceCancelled(msg.sender, nonce);
}
```