



Nucleus Security Review

Pashov Audit Group

Conducted by: Said, ast3ros, merlinboii

December 14th 2024 - December 16th 2024

Contents

1. About Pashov Audit Group	2
2. Disclaimer	2
3. Introduction	2
4. About Nucleus	2
5. Risk Classification	3
5.1. Impact	3
5.2. Likelihood	3
5.3. Action required for severity levels	4
6. Security Assessment Summary	4
7. Executive Summary	5
8. Findings	7
8.1. Medium Findings	7
[M-01] Bridge messages can be permanently lost	7
[M-02] depositAndBridge() fails when shareLockPeriod is active	8
[M-03] Custom hook not applied in _bridge() and _quote()	10
8.2. Low Findings	13
[L-01] The isAtomicRequestValid check is inconsistent with the solve function	13
[L-02] Lack of targetTeller validation	13
[L-03] _quote payload doesn't include messageId	13
[L-04] Risk of relying on InterchainSecurityModule	14
[L-05] Potential overestimation of assetsForWant passed to finishSolve()	15

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check 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. Introduction

A time-boxed security review of the **Ion-Protocol/nucleus-boring-vault** and **Ion-Protocol/nucleus-queues** repositories was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Nucleus

Ion is a lending built for staked and restaked assets. Borrowers can collateralize their yield-bearing staking assets to borrow WETH, and lenders can gain exposure to the boosted staking yield generated by borrower collateral. The scope for this audit consisted of two separate parts - one for integrating Hyperlane for a crosschain deposit contract, and one for facilitating solver based withdrawals akin to CoW protocol.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. 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.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hashes:

- fe84445ff7142d37ea2a88fa587a00c7515b4f13
- fbe7b5e1489139574ac8c2c3d19dc25fd53f86d9

fixes review commit hashes:

- fee3bafbb38b4587cd349ce0aed665e278a57490
- 6728bddf38bbb81ee362f958a2d94e18960273e6

Scope

The following smart contracts were in scope of the audit:

- `MultiChainHyperlaneTellerWithMultiAssetSupport`
- `CrossChainTellerBase`
- `MultiChainTellerBase`
- `TellerWithMultiAssetSupport`
- `AtomicQueueUCP`

7. Executive Summary

Over the course of the security review, Said, ast3ros, merlinboii engaged with Ion to review Nucleus. In this period of time a total of **8** issues were uncovered.

Protocol Summary

Protocol Name	Nucleus
Repository	https://github.com/Ion-Protocol/nucleus-queues
Date	December 14th 2024 - December 16th 2024
Protocol Type	Lending

Findings Count

Severity	Amount
Medium	3
Low	5
Total Findings	8

Summary of Findings

ID	Title	Severity	Status
[<u>M-01</u>]	Bridge messages can be permanently lost	Medium	Acknowledged
[<u>M-02</u>]	depositAndBridge() fails when shareLockPeriod is active	Medium	Acknowledged
[<u>M-03</u>]	Custom hook not applied in _bridge() and _quote()	Medium	Resolved
[<u>L-01</u>]	The isAtomicRequestValid check is inconsistent with the solve function	Low	Resolved
[<u>L-02</u>]	Lack of targetTeller validation	Low	Resolved
[<u>L-03</u>]	_quote payload doesn't include messageId	Low	Resolved
[<u>L-04</u>]	Risk of relying on InterchainSecurityModule	Low	Acknowledged
[<u>L-05</u>]	Potential overestimation of assetsForWant passed to finishSolve()	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] Bridge messages can be permanently lost

Severity

Impact: High

Likelihood: Low

Description

The `MultiChainHyperlaneTellerWithMultiAssetSupport` contract includes a pause mechanism that blocks all message handling through the `_beforeReceive` check in the `handle` function:

```
function handle
    (uint32 origin, bytes32 sender, bytes calldata payload) external payable {
    _beforeReceive();
    ...
}

function _beforeReceive() internal virtual {
    if (isPaused) revert TellerWithMultiAssetSupport__Paused();
}
```

This creates a vulnerability in the cross-chain bridge flow:

- User initiates a bridge transaction by calling `bridge` which burns their shares on the source chain
- Hyperlane relayer picks up the message and attempts delivery
- If the contract is paused during the relay period, message delivery will fail

Per Hyperlane docs, relayers do not guarantee infinite retries:

The
retry count of a message determines its next delivery attempt according to an exponential backoff.
Currently,
there is no fixed maximum number of retries after which the relayer will cease to attempt delivery.

<https://docs.hyperlane.xyz/docs/protocol/agents/relayer#the-submitter>

It results in:

- User's shares burned on the source chain
- No shares minted on the destination chain
- Funds effectively lost with no recovery mechanism

Recommendations

Handle the case where the `handle` transaction fails due to the paused state.

[M-02] `depositAndBridge()` fails when `shareLockPeriod` is active

Severity

Impact: Medium

Likelihood: Medium

Description

The `depositAndBridge` function attempts to perform a deposit followed immediately by a bridge operation, but this sequence fails when share locking is enabled due to contradictory security checks.

- `depositAndBridge` first calls `_afterPublicDeposit` which sets:
`shareUnlockTime[user] = block.timestamp + shareLockPeriod;`

```
function depositAndBridge(
    ERC20 depositAsset,
    uint256 depositAmount,
    uint256 minimumMint,
    BridgeData calldata data
)
    external
    payable
    requiresAuth
    nonReentrant
{
    ...
    _afterPublicDeposit
    //(msg.sender, depositAsset, depositAmount, shareAmount, shareLockPeriod); //
    bridge(shareAmount, data);
}
```

- It then immediately calls `bridge` which performs:

```
beforeTransfer(msg.sender);
```

```
function bridge(
    uint256 shareAmount,
    BridgeData calldata data
)
    public
    payable
    requiresAuth
    returns (bytes32 messageId)
{
    ...
    // Since shares are directly burned, call `beforeTransfer` to enforce
    // before transfer hooks.
    beforeTransfer
    //(msg.sender); // @audit revert because shareUnlockTime > block.timestamp
    ...
}
```

- The `beforeTransfer` check reverts if `shareUnlockTime > block.timestamp`:

```
function beforeTransfer(address from) public view {
    if
        (shareUnlockTime[from] > block.timestamp) revert TellerWithMultiAssetSupport
}
```

This leads to users cannot use the atomic `depositAndBridge` functionality when share locking is enabled. They must instead perform deposit and bridge as separate transactions with a waiting period in between.

Recommendations

- Add a bypass flag for the lock check during `depositAndBridge` and perform lock it in the destination chain.

[M-03] Custom hook not applied in

`_bridge()` and `_quote()`

Severity

Impact: High

Likelihood: Low

Description

The `_bridge()` and `_quote()` in the `MultiChainHyperlaneTellerWithMultiAssetSupport` contract fail to apply a custom post-dispatch hook set via the `setHook()`. Instead, they rely on the overloaded `Mailbox.dispatch()`, which defaults to using the Mailbox contract's defaultHook.

```
function _bridge(
    uint256shareAmount,
    BridgeDatacallldata
) internal override returns (bytes32 messageId)
--- SNIPPED ---
    mailbox.dispatch{ value: msg.value }(
        data.chainSelector, // must be `destinationDomain` on hyperlane
        msgRecipient, // must be the teller address left-padded to bytes32
        _payload,
        StandardHookMetadata.overrideGasLimit
        //(data.messageGas) // Sets the refund address to msg.sender, sets
        // `_msgValue`
        // to zero
    );
}
```

The `Mailbox.dispatch()` has three overloaded variants. In this case, the `_bridge()` invokes the version without specifying a custom `IPostDispatchHook`, causing the `Mailbox` contract to automatically apply with its `defaultHook`.

```
// hyperlane-monorepo/solidity/contracts/Mailbox.sol
function dispatch(
    uint32 destinationDomain,
    bytes32 recipientAddress,
    bytes calldata messageBody,
    bytes calldata hookMetadata
) external payable override returns (bytes32) {
    return
        dispatch(
            destinationDomain,
            recipientAddress,
            messageBody,
            hookMetadata,
@>            defaultHook
        );
}
```

Consequently, the `MultiChainHyperlaneTellerWithMultiAssetSupport` opts to use a custom post-dispatch hook instead of the `Mailbox` contract's default, the custom hook is not invoked during the dispatch process. This could result in incorrect or unintended post-dispatch handling.

```
// hyperlane-monorepo/solidity/contracts/Mailbox.sol
function dispatch(
    uint32 destinationDomain,
    bytes32 recipientAddress,
    bytes calldata messageBody,
    bytes calldata metadata,
    IPostDispatchHook hook
) public payable virtual returns (bytes32) {
    if (address(hook) == address(0)) {
        hook = defaultHook;
    }
    --- SNIPPED ---
    requiredHook.postDispatch{value: requiredValue}(metadata, message);
@>    hook.postDispatch{value: msg.value - requiredValue}(metadata, message);

    return id;
}
```

Note that this issue also arises with the `_quote()` that can lead to incorrect fee estimation.

Recommendations

Consider using the overloaded `dispatch()` method in the `Mailbox` contract that accepts a custom hook.

This approach will handle both cases where a hook is set and not set because it will also default to using the `defaultHook` if the protocol decides to leave the hook as the zero address.

Please note that the custom hooks should be compatible with the Hyperlane standard hook.

```
function _bridge(
  uint256shareAmount,
  BridgeDataacalldata
) internal override returns (bytes32 messageId)
--- SNIPPED ---
  mailbox.dispatch{ value: msg.value }(
    data.chainSelector, // must be `destinationDomain` on hyperlane
    msgRecipient, // must be the teller address left-padded to bytes32
    _payload,
    - StandardHookMetadata.overrideGasLimit
    - (data.messageGas) // Sets the refund address to msg.sender, sets
    - // `_msgValue`
    - // to zero
    + StandardHookMetadata.overrideGasLimit(data.messageGas),
    + hook
  );
}
```

8.2. Low Findings

[L-01] The `isAtomicRequestValid` check is inconsistent with the `solve` function

According to `isAtomicRequestValid`, when a user's `atomicPrice` is 0, it is considered an invalid atomic request. However, if other parameters (`deadline`, `offerAmount`) are configured correctly, the atomic request can still be solvable even when `atomicPrice` is 0. Consider either reverting the operation when `solve` is called and the user's `atomicPrice` is 0, or removing the `atomicPrice` is 0 check from `isAtomicRequestValid`.

[L-02] Lack of `targetTeller` validation

Inside `MultiChainTellerBase`, there are several operations to allow or stop receiving messages from a configured chain (`allowMessagesFrom` / `allowMessagesTo`). However, it does not verify if `targetTeller` is a non-empty address when `allowMessagesFrom` / `allowMessagesTo` is set to true. This could cause issues, as `mailbox.dispatch` will set `msgRecipient` to an empty address if misconfigured. Consider adding additional verification.

[L-03] `_quote` payload doesn't include `messageId`

`_quote` creates `_payload`, which consists of `shareAmount` and `destinationChainReceiver`, and passes it to `mailbox.quoteDispatch`.

```
function _quote(
    uint256shareAmount,
    BridgeDatacallldata
) internal view override returns (uint256) {
    bytes memory _payload = abi.encode
    (shareAmount, data.destinationChainReceiver);
    bytes32 msgRecipient = _addressToBytes32
    (selectorToChains[data.chainSelector].targetTeller);

    return mailbox.quoteDispatch(
        _payload, msgRecipient
    );
}
```

However, when the actual bridge is performed, the payload consists of `shareAmount`, `destinationChainReceiver`, and `messageId`.

```
function _bridge(
    uint256shareAmount,
    BridgeDatacallldata
) internal override returns (bytes32 messageId) {
    // ...

    bytes memory _payload = abi.encode
    (shareAmount, data.destinationChainReceiver, messageId);

    // Unlike L0 that has a built in peer check, this contract must
    // constrain the message recipient itself. We do this by our own
    // configuration.
    bytes32 msgRecipient = _addressToBytes32
    (selectorToChains[data.chainSelector].targetTeller);
    mailbox.dispatch{ value: msg.value }(
        data.chainSelector, // must be `destinationDomain` on hyperlane
        msgRecipient, // must be the teller address left-padded to bytes32
        _payload,
        StandardHookMetadata.overrideGasLimit
        // (data.messageGas) // Sets the refund address to msg.sender, sets
        // `_msgValue`
        // to zero
    );
}
```

Consider to also provide `messageId` inside `_quote`'s payload.

[L-04] Risk of relying on InterchainSecurityModule

Setting `interchainSecurityModule` to `address(0)` causes the protocol to use Hyperlane's default ISM, which can be changed by the Hyperlane team at any time. This creates a security risk without the protocol's control.

```
function setInterchainSecurityModule
(IInterchainSecurityModule _interchainSecurityModule) external requiresAuth {
    interchainSecurityModule = _interchainSecurityModule;
    emit SetInterChainSecurityModule(address(_interchainSecurityModule));
}
```

```
function setDefaultIsm(address _module) public onlyOwner {
    require(
        Address.isContract(_module),
        "Mailbox: default ISM not contract"
    );
    defaultIsm = IInterchainSecurityModule(_module);
    emit DefaultIsmSet(_module);
}
```

For example, if Hyperlane's team sets the default ISM to `NoopIsm`, any attacker could call `mailbox.process()` to mint unauthorized shares since `NoopIsm` performs no security verification.

`NoopIsm`: [link](#)

The likelihood is very low, however the protocol should monitor the use ISM or set a secure ISM itself.

[L-05] Potential overestimation of `assetsForWant` passed to `finishSolve()`

The `assetsForWant` amount passed to the external `finishSolve()` during solving the request can be overestimated compared to the actual amount of tokens pulled from the `solver`.

```
function solve(
    --- SNIPPED ---
)
    external
    nonReentrant
{
    if
        (!isApprovedSolveCaller[msg.sender]) revert AtomicQueue__UnapprovedSolveCaller(m
    uint8 offerDecimals = offer.decimals();
    uint256 assetsToOffer = _handleFirstLoop
        (offer, want, users, clearingPrice, solver);
    @> uint256 assetsForWant = _calculateAssetAmount
        (assetsToOffer, clearingPrice, offerDecimals);

    @> IAtomicSolver(solver).finishSolve
        (runData, msg.sender, offer, want, assetsToOffer, assetsForWant);

    _handleSecondLoop(offer, want, users, clearingPrice, solver, offerDecimals);
}
```



```

function _handleSecondLoop(
--- SNIPPED ---
)
    internal
{
    for (uint256 i = users.length; i > 0;) {
        --- SNIPPED ---

        uint256 assetsToUser = _calculateAssetAmount
            (request.offerAmount, clearingPrice, offerDecimals);
    @>    want.safeTransferFrom(solver, user, assetsToUser);
        --- SNIPPED ---
    }
}

```

In the `_handleSecondLoop()`, the want tokens are pulled from the solver using amounts derived from each user's `request.offerAmount`, which is processed using the `_calculateAssetAmount()`. The calculation employs a flooring multiplication-division operation, resulting in potential rounding down.

```

function _calculateAssetAmount(
--- SNIPPED ---
)
    internal
    pure
    returns (uint256)
{
    return clearingPrice.mulDivDown(offerAmount, 10 ** offerDecimals);
}

```

Consequently, as the `assetsForWant` is derived from the cumulative `assetsToOffer` values across all users but the actual token transfer amounts are rounded down for each user's request, this can lead to an overestimation of `assetsForWant` compared to the tokens actually transferred from the `solver`.

Note that this discrepancy of `assetsForWant` also arises with the `viewSolveMetaData()`

Consider using the sum of the actual `want` tokens that will be pulled for each user instead.