SHERLOCK

Security Review For Byzanlink

Collaborative Audit Prepared For: **Byzanlink**
Lead Security Expert(s): **pkqs90**
xiaoming90
Date Audited: **November 17 - November 19, 2025**

Introduction

Byzanlink V2 Vault is a non-custodial vault framework designed to support standardized on-chain asset management through the ERC-4626 tokenized vault standard, with ERC-2612 permit support. The system facilitates structured asset flows using both synchronous and asynchronous deposit and withdrawal mechanisms. The smart contracts operate with immutable logic and do not provide custodial control over user assets.

Scope

Repository: Byzanlink/ByzanlinkV2Vault

Audited Commit: 7b6dbba1b51606d00546e7bc9d629e0d28702f29

Final Commit: 5fec07143696ceb509dafce1c74c78daf4690f54

Files:

- contracts/adapters/nAlphaAdapter.sol
- contracts/ByzanlinkWithdrawalManager.sol
- contracts/libraries/doubleEndedQueue.sol
- contracts/libraries/Panic.sol
- contracts/VaultV2.sol

Final Commit Hash

5fec07143696ceb509dafce1c74c78daf4690f54

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
2	5	10

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Allocation cap tracking underflows when deallocating yield-bearing positions from nAlphaAdapter [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/6

Vulnerability Detail

VaultV2 tracks each adapter's allocation using the `caps[id].allocation` mapping. When `allocate()` or `deallocate()` is called, the adapter returns an `int256 change` value that represents the delta in the adapter's position. VaultV2 then updates the allocation:

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/VaultV2.sol#L588 https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/VaultV2.sol#L616

```
_caps.allocation = (int256(_caps.allocation) + change).toUint256();
```

The nAlphaAdapter incorrectly returns the absolute amount being allocated/deallocated instead of the actual `realAsset()` delta:

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/adapters/nAlphaAdapter.sol#L126 https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/adapters/nAlphaAdapter.sol#L201

```
    function allocate(
        bytes memory /* data */,
        uint256 assets,
        bytes4 /* selector */,
        address /* sender */
    ) external override onlyVault nonReentrant returns (bytes32[] memory, int256
    ↪  change) {
        ...
@>      return (ids(), int256(assets));
    }

    function deallocate(
        bytes memory,
        uint256 assets,
        bytes4,
        address
    ) external override onlyVault nonReentrant returns (bytes32[] memory, int256) {
        ...
        return (ids(), -int256(assets));
    }
```

A scenario is, if 100 USDC is allocated to nAlpha, then later it grows to 105 USDC, when we try to deallocate 105 USDC, the vaultv2's allocation track will underflow (100 - 105).

Impact

The protocol will revert when attempting to deallocate from nAlphaAdapter after yield accrual.

Recommendation

Use the implementation as seen in `MorphoVaultV1Adapter`, return the difference between old allocation and new `realAssets()`.

Discussion

Raghul2753

Thanks for highlighting this scenario.

Yes, this is a valid finding – when the allocated amount increases due to yield (e.g., 100 USDC growing to 105 USDC in nAlpha), attempting to directly deduct 105 from the recorded allocation would cause an underflow and revert.

This was a miss on our end. We will adopt the same approach as seen in MorphoVaultV1Adapter, and instead compute the delta as:

newRealAssets() - oldAllocation

and return that difference, rather than subtracting the full amount from the tracked allocation.

This ensures the allocation tracking remains safe and accurate even after yield accrues.

A fix will be added in the next commit.

Issue H-2: Withdrawal queue processing can be DoS'd with spam requests causing out-of-gas errors [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/7

Vulnerability Detail

Both `withdrawFundsFromAdapter()` and `processWithdrawals()` iterate through all pending withdrawal requests in unbounded loops:

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L170-L176

```
for (uint256 i = startIndex; i < currentQueueEnd ; i++) {
    bytes memory encodedData = withdrawalQueue._data[uint128(i)];
    WithdrawQueueStructs.WithdrawRequestData memory data =
    ↪  _decodeWithdrawData(encodedData);
    emit EventsLib.NalphaRequestItem(i, data.user, data.receiver, data.shares,
    ↪  data.expectedAssets, data.requestHash);
    totalShares += data.shares;
    count++;
}
```

Each user can submit up to `maxPendingPerUser` (default 5) withdrawal requests with no minimum amount. An attacker can create multiple addresses and spam thousands of 1 wei withdrawal requests. When the operator attempts to process these, the loop iterates through every request, eventually hitting the block gas limit and reverting.

Impact

Withdrawal processing becomes permanently blocked as the queue grows beyond the gas limit threshold. Legitimate users cannot withdraw funds.

Recommendation

1. Add a `batchSize` parameter to both functions to process withdrawals in bounded batches.
2. Add a minimum redemption limit.

Discussion

Raghul2753

We agree with the issue and are planning to address it, but we want feedback on the best approach before finalizing implementation.

Minimum Withdrawl Amount : For the minimum withdrawal amount, we want to balance accessibility with protection against spam attacks. Setting a low threshold such as 1 USDC keeps the system user-friendly, but still allows attackers to submit many tiny requests cheaply. Setting a slightly higher limit such as 5–10 USDC offers stronger protection, but may be seen as restrictive. Another approach is making the minimum withdrawal limit configurable so governance can adjust it over time as real usage data becomes available. From Defi apps, What would be your suggestion to keep the minimum amount, for prevention D'Dos attacks

Batch Size in Contract Regarding processing limits, one option is to store a batch size inside the contract and allow governance to update it. This ensures the processing loop always remains within a controlled limit, avoiding gas exhaustion while still being adjustable through governance decisions. We are planning to keep this Value as 1000. Is this value fine, or we have keeep any lower or higher values.

Sending Batch Size as Parameters The other option is to pass the batch size as a parameter to the withdrawal processing function. This gives the operator the flexibility to select how many requests to process in a single call, based on current gas conditions and queue growth.

We would like feedback on which approach is generally preferred in production DeFi systems and what minimum withdrawal limit is considered reasonable.

pkqs90

So since the current design requires everything handled *onchain*, hitting gas limit is inevitable. Especially during `processWithdrawals()` where each loop requires a ERC20 transfer (which is gas costly due to storage writes).

A more robust design would be something similar to Lido's withdrawQueue where 1) iterating requests in `withdrawFundsFromAdapter()` is handled *offchain*, and 2) users need to claim shares for the finalized withdrawals themselves instead of operators claiming for them. This way no looping is involved onchain.

Considering that would be a major change to the current design, I'd recommend the following:

1. Set 1 USDC as minimum withdrawal limit, but also make it updateable
2. Have operators manually pass in the amount of requests to process instead of hardcoding it, for both `withdrawFundsFromAdapter()` and `processWithdrawals()`
3. In `cancelWithdrawals()` add a feature to transfer the vault shares to treasury, to punish malicious users if they try to ddos the queue

Issue M-1: Async deallocations may fail when idle asset donations inflate withdrawal amounts beyond adapter's BoringVault shares [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/8

Vulnerability Detail

VaultV2's `realAssets()` calculation includes idle USDC held in both the vault itself and the nAlphaAdapter. When someone donates USDC to either VaultV2 or the nAlphaAdapter, the vault's asset/share ratio increases.

In `deallocateAsync()`, the adapter calculates how many nAlphaVault shares to redeem based on the withdrawal amount:

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/adapters/nAlphaAdapter.sol#L144

```
    function deallocateAsync(
        bytes memory /* data */,
        uint256 assets,
        bytes4 /* selector */,
        address sender
    ) external nonReentrant onlyVault {
        ...
@>      uint256 boringVaultShares = _assetsToShares(assets);
        ...
    }
```

Consider this scenario:

1. User deposits 100 USDC to VaultV2, which allocates to nAlphaAdapter
2. Adapter deposits into BoringVault, receiving 100 shares (1:1 ratio)
3. Someone donates 2 USDC idle assets to VaultV2 or nAlphaAdapter
4. VaultV2's total assets increase to 102 USDC, improving the share price
5. User withdraws, requesting `assets = 102` USDC worth
6. `deallocateAsync()` calculates `boringVaultShares = 102`
7. Adapter only holds 100 BoringVault shares

The subsequent `nAlphaAtomicQueue.updateAtomicRequest()` call fails because the adapter cannot transfer 102 shares when it only has 100.

Impact

User withdrawals may fail when idle asset donations inflate withdrawal amounts beyond the adapter's actual BoringVault share balance.

Recommendation

Change to: `uint256 boringVaultShares = min(boringVault.balanceOf(address(this)), _assetsToShares(assets));`

This is an edge case when nAlphaAdapter only has last amount of nAlphaShares left. In this case, if `boringVaultShares < boringVault.balanceOf(address(this))`, it should be sufficient to submit an atomicRequest for redeeming the `boringVault.balanceOf(address(this))` amount instead of `boringVaultShares`. After all nAlphaShares is redeemed to USDC, user can always complete withdrawal during `processWithdrawals()` phase.

Discussion

Raghul2753

We agree to this Issue . We will take the Minimum value of boring vault shares of the adapter or assetsToShares value, as last request would fail if the shares exceeds the current holding.

We have another question on keeping the idle assets in Adapter. Is it okay to keep some idle assets in Adapter, or we have to keep the adapter clean of not keeping any idle assets.

pkqs90

I think it's okay to keep some idle assets in the adapter as a buffer. This will also help mitigate the issue mentioned here https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/11.

Issue M-2: Blocklisted receiver address blocks entire withdrawal queue processing [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/10

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

The `processWithdrawals()` function processes withdrawal requests sequentially from the front of the queue. For each request, it transfers assets to the user-specified `receiver` address:

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L197

```
while (!withdrawalQueue.empty() && withdrawalQueue._begin < fundsWithdrawn) {
    bytes memory encodedData = withdrawalQueue.front();
    WithdrawQueueStructs.WithdrawRequestData memory data =
    ↪  _decodeWithdrawData(encodedData);
    uint256 assets = vault.previewRedeem(data.shares);

    bool isReady = asset.balanceOf(address(adapter)) >= assets;
    if (!isReady) break;

@>  uint256 assetsReceived = vault.redeem(data.shares, data.receiver,
↪   address(this));
    ...
}
```

USDC has a blocklist mechanism where certain addresses cannot receive transfers. If a user specifies a blocklisted address as their `receiver` when calling `requestWithdrawal()`, the `vault.redeem()` call will revert when USDC transfer fails. Since the queue processes in FIFO order, this malicious request blocks subsequent legitimate withdrawals until the owner manually cancels it via `cancelWithdrawals()`.

However, in the current implementation, for the withdrawals that are placed *before* the blocklisted receiver, they cannot be processed and must be cancelled as well.

Impact

A malicious user can DoS the withdrawal system by submitting a withdrawal request with a blocklisted receiver address.

Recommendation

Add a `withdrawCount` to the `processWithdrawals()` function to process up to only `withdrawCount` withdraws, so admins can still trigger the normal withdraws, and only cancel the pending one.

Discussion

Raghul2753

Hi @pkqs90 – thanks for bringing this issue to our attention.

This is a valid concern: since processWithdrawals() operates strictly in FIFO order, a withdrawal request targeting a USDC blocklisted address can cause vault.redeem() to revert and block subsequent legitimate withdrawals until manual intervention via cancelWithdrawals().

To address this, we are evaluating a few possible approaches:

1. use batch size parameter from Issue #7 one of the questions we raised was whether we can pass batchSize as a parameter to control how many withdrawals are processed in a single call. If that approach is implemented, we can reuse the same parameter here as well. This would ensure that processWithdrawals() only processes a limited number of requests per execution, allowing admins to continue processing other withdrawals instead of getting blocked by a single failing one.
2. Use both batch size and process count Another option is to introduce a dedicated limit (e.g., withdrawCount) in addition to batch size, giving more flexibility in how many withdrawals are executed per call. This could provide finer control to operators in different situations.
3. Pre-check for blacklisted receiver addresses USDC provides an isBlacklisted check. We could use this: During requestWithdrawal() to reject blocklisted receivers upfront, or During processWithdrawals() to detect such cases and automatically invalidate/remove the entry from the queue.

We will evaluate and implement the best option to ensure that a malicious or invalid receiver address cannot halt the withdrawal pipeline.

pkqs90

I think it's best to allow the operator to specify the amount of requests to process for both `processWithdrawals()` and `withdrawFundsFromAdapter()`. Best to make it a passable parameter for both functions.

For the blocklist example, consider this case, in the queue there are: `[7 normal requests, 1 blocklisted request, 10 normal requests]`, then admin can call `processWithdrawals(withdrawCount = 7)` to first process the 7 normal requests, cancel the 1 blocklisted request, and process the others.

Issue M-3: Withdrawal processing fails when VaultV2 share price increases between async deallocation and processing [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/11

Vulnerability Detail

The withdrawal system operates in two steps: `withdrawFundsFromAdapter()` initiates async deallocation, then `processWithdrawals()` executes the redemptions. The amount deallocated is calculated at initiation time:

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L180

```
if (totalShares > 0) {
    uint256 totalAssets = vault.convertToAssets(totalShares);
    fundsWithdrawn = newQueueEndIndex;
    vault.deallocateAsync(adapter, "", totalAssets);
}
```

During async processing (which can take days for nAlpha atomic queue resolution), VaultV2's share price can increase from: (1) nAlpha/USDC exchange rate appreciation, or (2) idle USDC donations to VaultV2 or the adapter. When `processWithdrawals()` executes, it recalculates based on the current price:

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L200-L202

```
uint256 assets = vault.previewRedeem(data.shares);

bool isReady = asset.balanceOf(address(adapter)) >= assets;
if (!isReady) break;
```

If share price increased from 1.00 to 1.05, the adapter only has assets for 1.00 per share but needs 1.05, causing `isReady = false` and halting all withdrawal processing.

Impact

Withdrawals become stuck whenever VaultV2's share price appreciates between deallocation request and processing. However, if there are >1 withdrawals in a batch, most likely only the last withdrawal will fail. If the batch only has one request, then it will completely block until additional deallocations are initiated.

Recommendation

1. Allocate some USDC as buffer within nAlphaAdapter to handle nAlpha/USDC natural price growth
2. Change the `isReady` check to `bool isReady = asset.balanceOf(address(vault)) + asset.balanceOf(address(adapter)) >= assets;`, since the idle USDC in vault can be used for redemption as well

Discussion

Raghul2753

We agree with the issue and the scenario described. When the share price increases between the async deallocation request and the actual processing, the assets available in the adapter may not be sufficient to redeem at the new price, which can cause isReady to fail and halt withdrawal execution.

1. Buffer Size Suggestion

We would like guidance on what would be a reasonable idle buffer percentage to maintain in the adapter so that withdrawals do not fail under normal price movement.Since the idle assets are included in NAV and do not grow (unlike the deployed capital), maintaining too large a buffer may reduce growth and APY. So we are looking for a balanced threshold that: Minimizes the chance of withdrawal failures due to price appreciation.Minimizes negative impact on yield/APY

If there are industry-standard heuristics (e.g., 0.5% – 2% idle buffer, dynamic buffer scaling, etc.), we would welcome input.

2. Updated isReady Logic

We will update the readiness check so it considers both: Idle assets in the adapter Idle assets in the vault

This allows redemption to proceed even if some liquidity is sitting in the vault instead of the adapter.

3. Liquidity Adapter Impact

Since we have Liquidity Adapter, Whenever capital flows in, Assets will be moved straight into nAlpha Vault. Idle liquidity will generally only exist in the vault or adapter due to donations or manual top-ups. Under normal operation, we expect almost all capital to be fully deployed, so this scenario should be rare–unless intentional buffer space is maintained.

We will implement the isReady logic change and seek guidance on a suitable idle buffer strategy to ensure stable operation without negatively impacting APY.

pkqs90

I think 1% buffer size should be good. Best to also make it adjustable for potential future changes.

Raghul2753

Hi @pkqs90 – I had one clarification regarding the allocation process.

For every allocation, should we maintain a configurable buffer within the adapter itself before assigning USDC, or is it better to manage the buffer manually from the Byzanlink side?

We are already planning to make the initial investment into the nAlpha vault. Along with that, we are considering donating around 10 USDC to the adapter as a small buffer.

From your perspective, which approach would be more appropriate?

Issue M-4: Lack of ability to limit the number of requests per batch leading to DOS [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/15

Vulnerability Detail

Noted that the `withdrawFundsFromAdapter()` function is intended to be executed every 24 hours. The issue is that the function batches all pending requests, and there is no option to limit the number of requests it batches.

If on a specific day, there is a large number of requests, attempting to batch all of them might result in an out-of-gas error and revert. In this case, none of the requests can be processed.

The same issue also applies to the `processWithdrawals()` function.

Impact

Withdrawal requests cannot be processed if there is a large number of legitimate pending requests.

Code Snippet

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/d076e40867cffc36102bc83fa4748b2e9313c093/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L160

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/d076e40867cffc36102bc83fa4748b2e9313c093/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L190

Tool Used

Manual Review

Recommendation

Consider adding an option that lets the operator define how many requests to batch each time to work around this.

Discussion

Raghul2753

We agree with the issue and are planning to address it, but we want feedback on the best approach before finalizing implementation.

Minimum Withdrawl Amount : For the minimum withdrawal amount, we want to balance accessibility with protection against spam attacks. Setting a low threshold such as 1 USDC keeps the system user-friendly, but still allows attackers to submit many tiny requests cheaply. Setting a slightly higher limit such as 5–10 USDC offers stronger protection, but may be seen as restrictive. Another approach is making the minimum withdrawal limit configurable so governance can adjust it over time as real usage data becomes available. From Defi apps, What would be your suggestion to keep the minimum amount, for prevention D'Dos attacks

Batch Size in Contract Regarding processing limits, one option is to store a batch size inside the contract and allow governance to update it. This ensures the processing loop always remains within a controlled limit, avoiding gas exhaustion while still being adjustable through governance decisions. We are planning to keep this Value as 1000. Is this value fine, or we have keeep any lower or higher values.

Sending Batch Size as Parameters The other option is to pass the batch size as a parameter to the withdrawal processing function. This gives the operator the flexibility to select how many requests to process in a single call, based on current gas conditions and queue growth.

As We got reply from #7 , We will pass the betch size as parameters to both the methods

Issue M-5: Withdrawal requests cannot be processed even if there is sufficient liquidity [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/16

Vulnerability Detail

The `isReady` only checks if there is sufficient liquidity (USDC) in the adapter contract.

However, after reviewing the code of the `VaultV2.exit()` function, the redemption can proceed as long as there is sufficient liquidity in the VaultV2 AND the adaptor. In Line 818 below, it will check if there is sufficient liquidity in the VaultV2. If not, it will attempt to fetch `assets - idleAssets` assets from the vault's liquidity adaptor to cover it.

```
File: VaultV2.sol
813:     /// @dev Internal function for withdraw and redeem.
814:     function exit(uint256 assets, uint256 shares, address receiver, address
 ↪  onBehalf) internal {
815:         require(canSendShares(onBehalf), ErrorsLib.CannotSendShares());
816:         require(canReceiveAssets(receiver), ErrorsLib.CannotReceiveAssets());
817:
818:         uint256 idleAssets = IERC20(asset).balanceOf(address(this));
819:         if (assets > idleAssets && liquidityAdapter != address(0)) {
820:             deallocateInternal(liquidityAdapter, liquidityData, assets -
 ↪  idleAssets);
821:         }
```

Thus, even if there is sufficient liquidity in the VaultV2 + adaptor for the redemption to proceed, the code will incorrectly assume there is a liquidity shortage because it only considers the liquidity in the adaptor.

```
File: ByzanlinkWithdrawalManager.sol
190:     function processWithdrawals(address adapter) external nonReentrant
 ↪  onlyOperator whenNotPaused {
191:         if (withdrawalQueue.empty()) revert ErrorsLib.QueueEmpty();
192:         if (withdrawalQueue._begin >= fundsWithdrawn) revert
 ↪  ErrorsLib.NoItemsSentToNAlpha();
193:
194:         uint256 processed = 0;
195:         uint256 totalDistributed = 0;
196:
197:         while (!withdrawalQueue.empty() && withdrawalQueue._begin <
 ↪  fundsWithdrawn) {
198:             bytes memory encodedData = withdrawalQueue.front();
199:             WithdrawQueueStructs.WithdrawRequestData memory data =
 ↪  _decodeWithdrawData(encodedData);
200:             uint256 assets = vault.previewRedeem(data.shares);
```

```
201:
202:             bool isReady = asset.balanceOf(address(adapter)) >= assets;
```

Impact

Withdrawal requests cannot be processed even if there is sufficient liquidity.

Code Snippet

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/d076e40867cffc36102bc83fa4748b2e9313c093/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L202

Tool Used

Manual Review

Recommendation

Consider implementing a more accurate check, as shown below.

```
- bool isReady = asset.balanceOf(address(adapter)) >= assets;
+ bool isReady = (asset.balanceOf(address(adapter)) +
 ↪  asset.balanceOf(address(vault))) >= assets;
```

Discussion

Raghul2753

As Discussed in Issue #11

We will update the isReady Variable to take the idle balance of adapter and the idle balance of Vault

Issue L-1: Withdrawal processing cannot handle distributed liquidity across multiple adapters [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/9

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

The `withdrawFundsFromAdapter()` function processes all pending withdrawal requests by calculating the total required assets and attempting to deallocate them from a single specified adapter:

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L160

```
function withdrawFundsFromAdapter(address adapter) external onlyOperator {
    ...
    for (uint256 i = startIndex; i < currentQueueEnd ; i++) {
        bytes memory encodedData = withdrawalQueue._data[uint128(i)];
        WithdrawQueueStructs.WithdrawRequestData memory data =
        ↪  _decodeWithdrawData(encodedData);
        totalShares += data.shares;
        count++;
    }

    if (totalShares > 0) {
        uint256 totalAssets = vault.convertToAssets(totalShares);
        vault.deallocateAsync(adapter, "", totalAssets);
        ...
    }
}
```

If VaultV2 uses multiple nAlphaAdapters with distributed allocations (e.g., Adapter A with 1000 USDC, Adapter B with 1000 USDC), and users queue 1500 USDC worth of withdrawals, calling `withdrawFundsFromAdapter()` will attempt to deallocate 1500 USDC from an adapter, which neither A or B has. This fails despite the protocol having 2000 USDC total liquidity across both adapters.

Note: Considering the first version only supports one adapter, this isn't an issue for now.

Impact

When liquidity is distributed across multiple adapters, withdrawal processing becomes impossible if queued requests exceed any single adapter's capacity, even when

aggregate protocol liquidity is sufficient. Users cannot withdraw funds despite available liquidity.

Recommendation

Either: (1) document that only a single adapter should be used, or (2) modify `withdrawFundsFromAdapter()` to accept multiple adapters and distribute deallocations proportionally based on each adapter's available liquidity.

Discussion

Raghul2753

Hi @pkqs90 – thanks for raising this openly.

You are correct. We initially explored supporting multiple adapters directly within the Withdrawal Manager. However, implementing on-chain allocation and deallocation logic across multiple adapters introduces significant complexity. To avoid this, we decided to keep the allocation handling off-chain for the current release.

With the current design, the Withdrawal Manager effectively supports only one adapter, and adding a second adapter would break the existing flow. We will document this clearly as a current limitation: the present Withdrawal Manager supports only a single adapter.

Looking ahead, our roadmap includes introducing a dedicated Liquidity Adapter layer that will internally manage distribution across multiple adapters. In that model:

1. Allocation requests will be split and routed to multiple adapters off-chain.
2. Deallocation logic will follow the same approach.
3. The Withdrawal Manager will still interact with only one Liquidity Adapter, keeping its interface unchanged.

This enhancement is part of our planned future improvements.

Issue L-2: Deposit fees from BoringVault are unfairly socialized to all VaultV2 shareholders [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/12

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

When users deposit into VaultV2, the vault allocates funds to the `liquidityAdapter` (nAlphaAdapter) if configured. The adapter then deposits assets into the underlying BoringVault via the Teller:

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/adapters/nAlphaAdapter.sol#L113-L118

```
uint256 boringVaultShares = teller.deposit(
    asset,
    assets,
    minimumSharesToMint
);
```

While nAlpha currently don't charge deposit fees, some BoringVault implementations (e.g., Seven Labs' TellerWithMultiAssetSupport) do charge deposit premiums.

If a deposit fee exists, the actual BoringVault shares received will be worth less than the deposited amount. However, VaultV2 mints shares to the depositor based on the full deposit amount before the fee. When `accrueInterestView()` is called, it sums `adapter.realAssets()` which reflects the post-fee value, causing an immediate loss that is socialized across all vault shareholders rather than being borne solely by the depositor.

This issue was previously identified as M-04 in the Morpho Vault V2 Blackthorn audit.

Impact

No impact for now, because nAlpha does not charge deposit fees. But this is something to look over for future integrations. If BoringVault charges deposit fees, each deposit unfairly dilutes existing shareholders rather than having the depositor bear their own deposit costs.

Recommendation

N/A

Discussion

Raghul2753

Hi @pkqs90 – thanks for bringing this up.

Since nAlpha currently has no deposit fee, this issue does not affect our present integration. However, we will document the problem and take it into account when integrating with future vaults that do apply deposit fees.

Just to confirm our understanding for future support: Each adapter should be able to define its own deposit fee configuration.

1. When a user deposits USDC, the adapter should:
2. Read the configured deposit fee rate
3. Subtract the fee from the input USDC
4. Deposit only the post-fee amount into the underlying vault
5. Mint shares based on the reduced amount

This ensures that VaultV2 tracks the actual deployable capital and that share minting reflects the net amount after fees.

Please confirm if this interpretation is correct.

pkqs90

I think ideally to support deposit fees, vault v2 has to stop supporting ”default liquidity adapters”, or the adapter logic would be too complicated, i.e. the adapter has to differentiate between a ”user deposit + liquidity allocate” vs. ”operator triggered liquidity allocate”.

The key is the two transactions: 1) user depositing USDC into vault v2, and 2) USDC being allocated to adapter, should not be in 1 transaction.

Issue L-3: Instant withdrawals incorrectly validate treasury balance including fees instead of actual transfer amount [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/13

Vulnerability Detail

The `requestInstantWithdrawal()` function allows users to instantly redeem VaultV2 shares by receiving assets directly from the treasury wallet, with a fee deducted. The function calculates the user's expected assets, applies a fee, and transfers the post-fee amount:

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/main/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L71

```
uint256 expectedAssets = vault.convertToAssets(shares);
uint256 feeAmount = expectedAssets.mulDivDown(instantWithdrawalFeeBps, BPSUNIT);
assetsReceived = expectedAssets - feeAmount;

// Check treasury has enough assets
if (expectedAssets > asset.balanceOf(treasuryWallet)) revert
 ↪  ErrorsLib.InsufficientTreasuryBalance();

// Transfer assets from treasury to receiver (after fee deduction)
if (!asset.transferFrom(treasuryWallet, receiver, assetsReceived)) revert
 ↪  ErrorsLib.AssetTransferFailed();
```

The balance check validates that the treasury has `expectedAssets`, but only `assetsReceived` is actually transferred. If the treasury balance is between `assetsReceived` and `expectedAssets`, the transaction reverts unnecessarily despite having sufficient funds to fulfill the withdrawal.

Impact

Instant withdrawals may revert when the treasury has enough assets.

Recommendation

Change the balance check to validate against the actual transfer amount:

```
if (assetsReceived > asset.balanceOf(treasuryWallet)) revert
 ↪  ErrorsLib.InsufficientTreasuryBalance();
```

Discussion

Raghul2753

Hi @pkqs90

It is a Miss from our End, previously we were not keeping the additional bps for the Instant Withdrawl

Thanks for Bringing this out. We will Update the code for this issue

Issue L-4: Multiple request withdrawals might have the same hash. [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/14

Vulnerability Detail

The value of `queueIndex` is retrieved from the withdrawal queue length (`withdrawalQueue.length()`). This variable is part of the input used to generate the request hash.

However, this might not produce a unique hash in an edge case.

Bob submits two (2) separate `requestWithdrawal` transactions to the mempool at the same time, or on two different times (T1 and T2).

```
TX1 - requestWithdrawal(100, Bob)
TX2 - requestWithdrawal(100, Bob)
```

Assume that in Block 1000 (blk.timestamp = 1000), the withdrawal queue length is 50. Both TX1 and TX2 get executed in Block 1000, and the transactions are ordered in the following manner:

1. TX1 - requestWithdrawal(100, Bob) - Hash input => [Bob, Bob, 100, T1000, 50]. The withdrawal queue length is incremented to 51.
2. `processWithdrawals()` TX is executed, and one request is processed. Thus, the Withdrawal Queue is reduced by one, and its length becomes 50 again.
3. TX2 - requestWithdrawal(100, Bob) - Hash input => [Bob, Bob, 100, T1000, 50]. The withdrawal queue length is incremented to 51.

Thus, there will be two request withdrawals with the same hash.

Impact

Multiple request withdrawals might share the same hash, leading to issues when referencing the request off-chain or on-chain.

Code Snippet

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/d076e40867cffc36102bc83fa4748b2e9313c093/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L84

Tool Used

Manual Review

Recommendation

Consider using a unique nonce variable that always increments by one, instead of relying on `withdrawalQueue.length()`, whose length might increase or decrease within a single block.

Discussion

Raghul2753

Hi @xiaoming9090

Yes We agree to this scenarios. We took reference from other Vault existing in the MArket and kept this. But yes there is a possiblity that it will have the same hash.

We will introduce the nonce while generating the hash

Issue L-5: Additional validation check should be implemented [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/17

Vulnerability Detail

It was observed that an arbitrary adaptor address can be passed into the `withdrawFundsFromAdapter()` and `processWithdrawals()` functions.

```
File: ByzanlinkWithdrawalManager.sol
160:     function withdrawFundsFromAdapter(address adapter) external onlyOperator
 ↪  nonReentrant whenNotPaused {
161:         if (withdrawalQueue.empty()) revert ErrorsLib.QueueEmpty();
162:
```

```
File: ByzanlinkWithdrawalManager.sol
190:     function processWithdrawals(address adapter) external nonReentrant
 ↪  onlyOperator whenNotPaused {
191:         if (withdrawalQueue.empty()) revert ErrorsLib.QueueEmpty();
192:         if (withdrawalQueue._begin >= fundsWithdrawn) revert
 ↪  ErrorsLib.NoItemsSentToNAlpha();
```

Impact

Informational. If an operator passes in the wrong address, it might lead to an unexpected result.

Code Snippet

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/d076e40867cffc36102bc83fa4748b2e9313c093/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L190

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/d076e40867cffc36102bc83fa4748b2e9313c093/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L160

Tool Used

Manual Review

Recommendation

Consider checking whether the `adaptor` address passed in is an authorized adaptor. This can be done by relying on the `VaultV2`'s public `isAdapter` mapping.

Discussion

Raghul2753

Hi @xiaoming9090

Yes Passing Wrong adapter in might cause issues. We will implement the Vault::isAdapter check in both processwithdrawls and withdrawFundsFromAdapter to mitigate any issues

Issue L-6: Hardcoded expiry duration [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/18

Vulnerability Detail

It was observed that the expiry duration is hardcoded to 10 days.

```
File: nAlphaAdapter.sol
135:     function deallocateAsync(
..SNIP..
150:         // Create withdrawal request
151:         uint256 requestId = nextRequestId++;
152:
153:         // Create atomic request in nAlpha queue
154:         IAtomicQueue.AtomicRequest memory request =
↪  IAtomicQueue.AtomicRequest({
155:             deadline: uint64(block.timestamp + 10 days),
156:             atomicPrice: uint88(currentPrice),
157:             offerAmount: uint96(boringVaultShares),
158:             inSolve: false
159:         });
```

Impact

Market conditions may change, causing atomic requests to be resolved faster or slower. However, since the expiry duration is hardcoded, there is no way to adjust the expiry duration of the atomic request to align with the current market conditions.

Code Snippet

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/d076e40867cffc36102bc83fa4748b2e9313c093/ByzanlinkV2Vault/contracts/adapters/nAlphaAdapter.sol#L155

Tool Used

Manual Review

Recommendation

Instead of hardcoding the expiry duration to 10 days, consider allowing the admin to modify it,

Discussion

Raghul2753

Hi @xiaoming9090

Thanks for Pointing out this Issue.

We will Keep it as variable and Create method to modify this Variable through Owner of nalpha Adapter

Issue L-7: Residual allowance if requests are cancelled [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/19

Vulnerability Detail

It was observed that the allowance is granted to the nAlpha vault whenever a new atomic request is submitted. This is required because the nAlpha will pull the nAlpha vault shares from the adaptor if the atomic request is solved.

```
File: nAlphaAdapter.sol
135:     function deallocateAsync(
..SNIP..
152:
153:         // Create atomic request in nAlpha queue
154:         IAtomicQueue.AtomicRequest memory request =
↪  IAtomicQueue.AtomicRequest({
155:             deadline: uint64(block.timestamp + 10 days),
156:             atomicPrice: uint88(currentPrice),
157:             offerAmount: uint96(boringVaultShares),
158:             inSolve: false
159:         });
160:
161:         IERC20(address(boringVault)).safeIncreaseAllowance(
162:             address(nAlphaAtomicQueue),
163:             uint256(boringVaultShares)
164:         );
```

However, there is no guarantee that all atomic requests will be solved. If the atomic request is cancelled or expired, the operator will call `ByzanlinkWithdrawalManager.cancelWithdrawals()` function. However, the allowance granted to nAlpha vault earlier is not revoked. Thus, it might end up in a situation where there is excess allowance granted to the nAlpha vault.

Impact

An excess allowance will be granted to the nAlpha vault.

Code Snippet

Tool Used

Manual Review

Recommendation

Consider revoking the unused allowance when the withdrawal requests are cancelled.

Discussion

Raghul2753

Hi @xiaoming9090

Thanks for Bringing this issue Out. We will Write helper functions in Vault and Adapter to Revoke the Allowance of nAlpha shares to Atomic Queue, whenever we cancle the withdrawls from the ByzanlinkWithdrawManager

Raghul2753

Hi @xiaoming9090

Need a suggestion on Handling other use cases.

For Instance, if we are going to cancel the Withdrawl request because of USDC Blocklisted Receiver address, where in case the requests are already processed in nalpha Side. In this case, We cant revoke the allowance, since the funds are already processed.

Solution: In the nalphaAdapter, we can check the Atomic request and only if it is invalid, we are revoking the Necessary allowance from the Atomic Queue.

So Revoking allowance wont works for Blacklisted address issue and works only for Expired atomic requests

Issue L-8: Atomic request might not be solved at the current price [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/20

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

It was observed that the adaptor submits an atomic request with the atomic price set to the current price.

However, the issue is that it is unlikely that this request will be solved at the current price, as there is no incentive for the solvers to do so. Some incentives are needed (e.g., a discount) to compensate the solvers for the gas cost, liquidity provisioning, and risk taken.

```
File: nAlphaAdapter.sol
135:     function deallocateAsync(
..SNIP..
148:         uint256 currentPrice = accountant.getRateInQuoteSafe(asset);
149:
150:         // Create withdrawal request
151:         uint256 requestId = nextRequestId++;
152:
153:         // Create atomic request in nAlpha queue
154:         IAtomicQueue.AtomicRequest memory request =
↪  IAtomicQueue.AtomicRequest({
155:             deadline: uint64(block.timestamp + 10 days),
156:             atomicPrice: uint88(currentPrice),
157:             offerAmount: uint96(boringVaultShares),
158:             inSolve: false
159:         });
```

Impact

Withdrawal requests cannot be processed if the atomic requests are not solved.

Code Snippet

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/d076e40867cffc36102bc83fa4748b2e9313c093/ByzanlinkV2Vault/contracts/adapters/nAlphaAdapter.sol#L148

Tool Used

Manual Review

Recommendation

Reconsider if there is any incentive for the current solvers to solve the atomic requests at the current price. Alternatively, the protocol should consider introducing some incentive. The `AtomicQueue` contract includes a safeUpdateAtomicRequest() function that allows the submitter to specify the discount that may be used.

Discussion

Raghul2753

In the nalpha Atomic queue, there is no Such method and we will discuss with nalpha regarding this issue

Issue L-9: Received assets might be lower than the `expectedAssets` [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/21

Vulnerability Detail

Assume that the current price of the VaultV2 share is 1.10 at T1.

At T1, a user attempts to withdraw 100 vault shares via the `ByzanlinkWithdrawalManager.requestWithdrawal()` function, and an event records the `expectedAssets` to be 110 assets (100 * 1.10). Similarly, the `encodedData` in Line 95 below also recorded the `expectedAssets` to be 110.

It takes some time for the withdrawal request to be processed. During this period, the price of vault shares might fall. Thus, it's possible that the amount of assets the user will receive at the end is less than the `expectedAssets` (110).

```
File: ByzanlinkWithdrawalManager.sol
071:     function requestWithdrawal(
072:         uint256 shares,
073:         address receiver
074:     ) external nonReentrant whenNotPaused returns (uint256 queueIndex) {
075:         if (receiver == address(0)) revert ErrorsLib.InvalidReceiver();
076:         if (userPending[msg.sender] >= maxPendingPerUser) revert
↪  ErrorsLib.TooManyPending();
077:
078:         uint256 expectedAssets = vault.convertToAssets(shares);
079:         if (expectedAssets == 0) revert ErrorsLib.InvalidConversion();
080:
081:         // Transfer shares from user to this contract
082:         if (!vault.transferFrom(msg.sender, address(this), shares)) revert
↪  ErrorsLib.ShareTransferFailed();
083:
084:         queueIndex = withdrawalQueue.length();
085:
086:         bytes32 requestHash = keccak256(
087:             abi.encode(msg.sender, receiver, shares, block.timestamp,
↪  queueIndex)
088:         );
089:
090:         bytes memory encodedData = abi.encode(
091:             WithdrawQueueStructs.WithdrawRequestData({
092:                 user: msg.sender,
093:                 receiver: receiver,
094:                 shares: uint128(shares),
095:                 expectedAssets: uint128(expectedAssets),
096:                 requestTimestamp: uint64(block.timestamp),
```

```
097:                 requestHash: requestHash
098:             })
099:         );
100:
101:         withdrawalQueue.pushBack(encodedData);
102:         userPending[msg.sender]++;
103:
104:         emit EventsLib.WithdrawalRequested(
105:             queueIndex,
106:             msg.sender,
107:             receiver,
108:             shares,
109:             expectedAssets,
110:             requestHash,
111:             block.timestamp
112:         );
113:     }
```

Impact

Users might expect to receive 110 assets, but in reality, there is no guarantee, as the share price might fall while the request is being processed.

Code Snippet

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/d076e40867cffc36102bc83fa4748b2e9313c093/ByzanlinkV2Vault/contracts/ByzanlinkWithdrawalManager.sol#L78

Tool Used

Manual Review

Recommendation

Ensure that this behavior is documented so that no one has the wrong expectations or assumptions about the actual amount of assets they will receive at the end.

Discussion

Raghul2753

Hi @xiaoming9090

Thanks for Pointing this out

This is already in our Radar and we will clearly document and indicate users that, expected assets would be based on the current price. Since. we have 10 day cooldown period, Actual assets will be based on the price on the Settlement Date. Whenever, we will get the Liquidity, Users fund will be settled. It will be based on the price on the day of settlement

Issue L-10: Malicious solvers can steal funds from `VaultV2` [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/issues/23

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

Assume that the nAlpha adaptor holds a total of 200 nAlpha vault shares, which are worth 5 USDC each. Thus, the total assets of the VaultV2 is 1000 USDC (200 *5). The total share/supply of VaultV2 is 1000 shares. Thus, the price of each VaultV2 share is 1.0 now.

The AtomicQueue.solve() function implemented the logic for the solver to solve the atomic request submitted by the nAlpha adaptor. In Line 325 below, the offer asset (nAlpha vault shares) will be transferred from the nAlpha adaptor to the solver address. Assume that 100 nAlpha vault shares have been transferred from the adaptor to the solver.

Then, in Line 328, it will pass the execution control to the solver address, which might be a smart contract.

An important point to note is that, at this point, the nAlpha adaptor holds only 100 nAlpha vault shares (50% of the initial amount). The issue is that a malicious solver who gains execution control and exploits it to mint a new VaultV2 share at a low price, as the current price of the VaultV2 share is 0.5 USDC. Assume that the malicious solvers mint 1000 VaultV2 shares and paid 500 USDC.

The solver will then process the atomic request as usual and transfer 500 USD to the adaptor in exchange for 100 nAlpha vault shares. At this point, the adaptor holds:

1. 500 USDC
2. 100 nAlpha vault shares (worth 500 USDC)

The total assets of the VaultV2 is 1000 USDC, and the total supply is 1500 (1000 + 500). Thus, the price per share is 0.6666 now. The malicious solver can then redeem all their 1000 VaultV2 shares, and earn a profit of around 166.6666 USD (666.666 - 500) at the expense of the existing holder of VaultV2 shares.

https://github.com/Se7en-Seas/boring-vault/blob/4538ccf7706290e38b739e939854897c0f81f6de/src/atomic-queue/AtomicQueue.sol#L299

```
File: AtomicQueue.sol
299:     function solve(ERC20 offer, ERC20 want, address[] calldata users, bytes
↪  calldata runData, address solver)
300:         external
301:         nonReentrant
302:         requiresAuth
303:     {
```

```
304:         if (isPaused) revert AtomicQueue__Paused();
305:
306:         // Save offer asset decimals.
307:         uint8 offerDecimals = offer.decimals();
308:
309:         uint256 assetsToOffer;
310:         uint256 assetsForWant;
311:         for (uint256 i; i < users.length; ++i) {
312:             AtomicRequest storage request =
↪  userAtomicRequest[users[i]][offer][want];
313:
314:             if (request.inSolve) revert AtomicQueue__UserRepeated(users[i]);
315:             if (block.timestamp > request.deadline) revert
↪  AtomicQueue__RequestDeadlineExceeded(users[i]);
316:             if (request.offerAmount == 0) revert
↪  AtomicQueue__ZeroOfferAmount(users[i]);
317:
318:             // User gets whatever their atomic price * offerAmount is.
319:             assetsForWant += _calculateAssetAmount(request.offerAmount,
↪  request.atomicPrice, offerDecimals);
320:
321:             // If all checks above passed, the users request is valid and
↪  should be fulfilled.
322:             assetsToOffer += request.offerAmount;
323:             request.inSolve = true;
324:             // Transfer shares from user to solver.
325:             offer.safeTransferFrom(users[i], solver, request.offerAmount);
326:         }
327:
328:         IAtomicSolver(solver).finishSolve(runData, msg.sender, offer, want,
↪  assetsToOffer, assetsForWant);
329:
330:         for (uint256 i; i < users.length; ++i) {
331:             AtomicRequest storage request =
↪  userAtomicRequest[users[i]][offer][want];
332:
333:             if (request.inSolve) {
334:                 // We know that the minimum price and deadline arguments are
↪  satisfied since this can only be true if they were.
335:
336:                 // Send user their share of assets.
337:                 uint256 assetsToUser =
↪  _calculateAssetAmount(request.offerAmount, request.atomicPrice, offerDecimals);
338:
339:                 want.safeTransferFrom(solver, users[i], assetsToUser);
..SNIP..
345:                 // Set shares to withdraw to 0.
346:                 request.offerAmount = 0;
347:                 request.inSolve = false;
348:             } else {
```

```
349:                 revert AtomicQueue__UserNotInSolve(users[i]);
350:             }
351:         }
352:     }
```

Impact

Malicious solvers can steal funds from the VaultV2.

Code Snippet

https://github.com/sherlock-audit/2025-11-byzanlink-nov-17th/blob/d076e40867cffc36102bc83fa4748b2e9313c093/ByzanlinkV2Vault/contracts/VaultV2.sol#L184

Tool Used

Manual Review

Recommendation

Although the AtomicQueue.solve() function is permissioned, it is still essential to ascertain the trustworthiness of the solvers, as they can technically steal funds from VaultV2. Proper due diligence on these solvers is necessary.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.