hexens x logarithm

Security Review Report for Logarithm

July 2025

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the security review for the Metavault of Logarithm Labs. The Metavault implements a more user friendly vault to interact with LogarithmVaults by handling deposits, withdrawals and allocations, while also supporting ERC4626.

Our security assessment was a full review of the scope, spanning a total of 1 week.

During our audit, we identified 1 critical severity vulnerability, which could have cause the share rate to be manipulated and principal assets to be extracted.

We have also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

3. Security Review Details

▪ Review Led by

Kasper Zwijsen, Head of Audits

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/Logarithm-Labs/logarithm_metavault/tree/98f7148a191dcd4e8f2664021f173dc926bb24cb

The issues described in this report were fixed in the following commit:

🔗 https://github.com/Logarithm-Labs/logarithm_metavault

📌 Commit: `1bdf8d0009d5831d34f04a0275dcd86d4df09df4`

▪ Changelog

22 July 2025	Audit start
30 July 2025	Initial report
01 August 2025	Revision received
05 August 2025	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

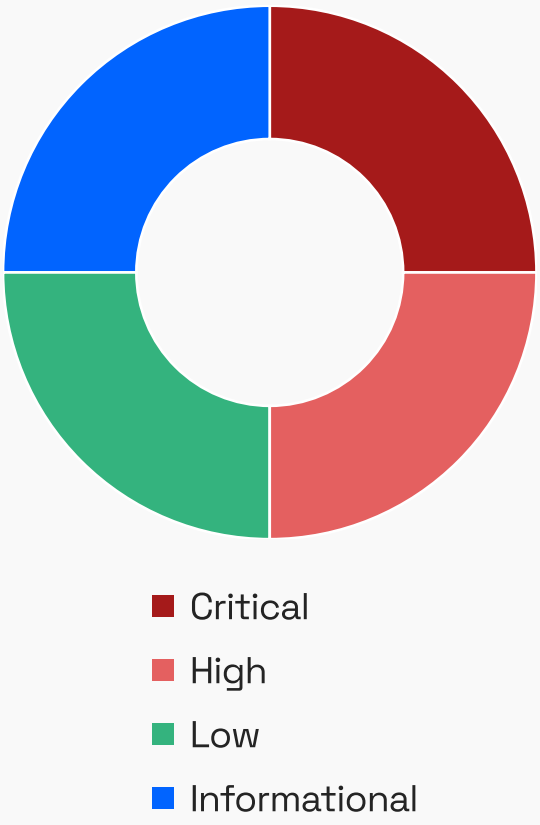
- **Issue Symbolic Codes**

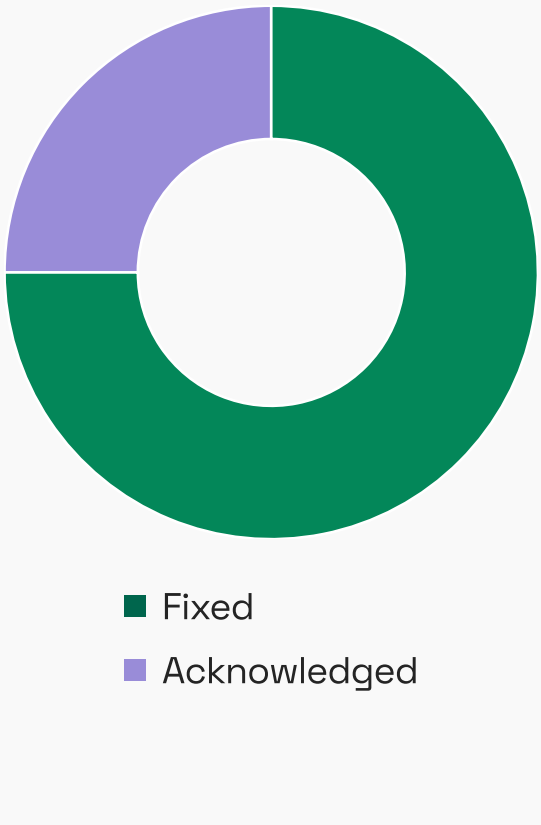
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	1
High	1
Medium	0
Low	1
Informational	1
Total:	**4**





6. Weaknesses

This section contains the list of discovered weaknesses.

LOGLAB3-4 | Incorrect totalAssets Calculation Due to Claimed-but-Unremoved WithdrawKeys

Fixed

Severity:	Probability:	Impact:
Critical	Likely	Critical

Path:

src/MetaVault.sol:claimAllocations#L448-L477

Description:

The MetaVault requests withdrawals from the LogarithmVault and keeps track of the withdrawal keys.

However, when a withdrawal is claimed on the LogarithmVault by a 3rd party, before the **claimAllocations** function is executed, that **withdrawKey** is never removed from the allocation list.

As a result, the **metaVault** continues to count the **requestedAssets** for that already-claimed key, leading to permanent inflation of **totalAssets()**. This inflates the share price and causes long-term loss of funds for honest vault users, as they receive fewer assets per share than they should.

In detail, the **claimAllocations** function in the metaVault, tries to claim a withdrawal from the **targetVault**. The **claim** function is public, anyone can call it to release the funds of any withdrawKey to its intended receiver.

The issue arises when someone else calls **claim** before **claimAllocations** does, then the **metaVault** will think the withdrawal is not yet ready and won't remove the **withdrawKey** because **isClaimable** with that **withdrawKey** on the target vault returns false because its already claimed.

```
function claimAllocations() public {
--- Snip ----
      if (ILogarithmVault(claimableVault).isClaimable(withdrawKey)) {
          ILogarithmVault(claimableVault).claim(withdrawKey);
          $.allocationWithdrawKeys[claimableVault].remove(withdrawKey);
      } else {
          allClaimed = false;
      }
```

If all **withdrawKeys** are successfully claimed and removed, then the vault is removed from the list:

```
if (allClaimed) {
    $.claimableVaults.remove(claimableVault);
}
```

But if any **withdrawKey** was already claimed externally, it won't be removed, which causes the vault to remain in **claimableVaults** even though it should be removed.

With this in mind **totalAssets** gets the requested and claimable values from unclaimed keys (and these claimed keys on target vault) and adds the assets balance and allocated balances

```
function totalAssets() public view override returns (uint256) {
    uint256 assetBalance = IERC20(asset()).balanceOf(address(this));
    (uint256 requestedAssets, uint256 claimableAssets) =
getWithdrawalsFromAllocation();
    (, uint256 assets) =
        (assetBalance + allocatedAssets() + requestedAssets +
claimableAssets).trySub(assetsToClaim());
    return assets;
}
```

So since the last withdraw key was claimed, increasing the **assetBalance**, but not removing the key it will add the **requestedAssets** of that "unclaimed" key

```
function getWithdrawalsFromAllocation() public view returns (uint256
requestedAssets, uint256 claimableAssets) {
... SNIP ...
                uint256 assets =
ILogarithmVault(claimableVault).withdrawRequests(withdrawKey).requestedAssets;

                if (ILogarithmVault(claimableVault).isClaimable(withdrawKey)) {
                    unchecked {
                        claimableAssets += assets;
                    }
                } else {
                    unchecked {
                        requestedAssets += assets;
                    }
                }
```

It adds the **requestedAssets** even if the withdrawal was already claimed. For example, if the target vault has already processed the claim, **isClaimable** will return **false**. As a result, it incorrectly adds the amount to **requestedAssets** even though it's no longer "requested" because it has already been received.

For example:
Initial Withdraw Request:
- A withdraw request for **1e18** is made on the target vault.
At this point in the metaVault:

```
assetBalance       = 0
requestedAssets    = 1e18
claimableAssets    = 0
totalAssets        = 1e18
```

Normal Flow (No Claim Made on the target vault):
the metaVault claims the assets.

```
assetBalance       = 1e18
requestedAssets    = 0
claimableAssets    = 0
totalAssets        = 1e18
```

Irregular Flow (Claim Was Made on Target Vault):

```
assetBalance       = 1e18
requestedAssets    = 1e18
claimableAssets    = 0
totalAssets        = 2e18
```

This is incorrect. The **totalAssets** should remain **1e18**

Since this key can never be removed it has long lasting effect on the share rate of the vault.

Remediation:

Add a check in **getWithdrawalsFromAllocation**:

```
WithdrawRequest memory req =
ILogarithmVault(claimableVault).withdrawRequests(withdrawKey);
if (req.isClaimed) continue; // So it does not count it as request/claimable
asset
```

And in **claimAllocations** another check to remove the withdraw key from the metaVault side if its **isClaimed** = true.

LOGLAB3-1 | MetaVault Improperly Handles Last Redeem Claim Shortfalls

Acknowledged

Severity:	Probability:	Impact:
High	Likely	High

Path:

src/MetaVault.sol:claimAllocations#L448-L477

Description:

The MetaVault interacts with underlying vaults (e.g. LogarithmVaults) to allocate assets for deployment and to request withdrawals of those allocations. The function **claimAllocations** allows for permissionless claiming of those withdrawal requests from the LogarithmVault, however it fails to take into account an edge case in claiming. If the claim is the last redeem in the target vault it has a different workflow, where it could reduce/increase the funds based on the shortfall: **managed_basis/vault/LogarithmVault.sol#L554-L586**

In the case of shortfall, the claimed amount will be less than expected, but MetaVault automatically accepts the claimed amount.

As a result, it causes the share rate to suddenly drop for the full amount of the shortfall. Any pending user withdrawals were made at the share rate before this claim, so if no further deposits are made, it would case a revert because the claim function tries to send the full withdraw request amount, even though it actually received less due to a shortfall:

```
function claim(bytes32 withdrawKey) public returns (uint256) {
.... Snip ....
@=> IERC20(asset()).safeTransfer(withdrawRequest.receiver,
withdrawRequest.requestedAssets);
```

On the other hand, if there are future deposits, the missing assets of the pending user withdrawal requests would be filled with other users' principal assets from future deposits. This might resolve itself, but in the case of large withdrawals and shortfall, it could cause significant accounting issues.

```
    function claimAllocations() public {
        MetaVaultStorage storage $ = _getMetaVaultStorage();
        address[] memory _claimableVaults = claimableVaults();
        uint256 len = _claimableVaults.length;
        for (uint256 i; i < len;) {
            address claimableVault = _claimableVaults[i];
            bytes32[] memory _allocationWithdrawKeys =
allocationWithdrawKeys(claimableVault);
            uint256 keyLen = _allocationWithdrawKeys.length;
            bool allClaimed = true;
            for (uint256 j; j < keyLen;) {
                bytes32 withdrawKey = _allocationWithdrawKeys[j];
                if (ILogarithmVault(claimableVault).isClaimable(withdrawKey)) {
                    ILogarithmVault(claimableVault).claim(withdrawKey);

$.allocationWithdrawKeys[claimableVault].remove(withdrawKey);
                } else {
                    allClaimed = false;
                }
                unchecked {
                    ++j;
                }
            }

            if (allClaimed) {
                $.claimableVaults.remove(claimableVault);
            }
            unchecked {
                ++i;
            }
        }
    }
```

Remediation:

The function **MetaVault:claim** that processes user withdrawal requests should take the current share rate into account and take the minimum between the share rate at request and the current share rate and use that to fulfil the request. This is the same approach as other liquid staking services where slashing/losses are possible, such as Lido.

Commentary from the client:

*"The **LogarithmVault** is designed so that users receive exactly the amount of assets they requested to withdraw. However, when there are insufficient funds to fulfill all withdrawal requests, the protocol itself becomes the final claimant to handle any remaining shortfall. To ensure this mechanism works properly, the team deposits initial funds before the vault launches.*

This behavior is by design and acknowledged as acceptable for the current implementation."

LOGLAB3-5 | Fee and Hurdle Rate Changes Apply to the Past

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

managed_basis/src/vault/ManagedVault.sol:setFeeInfos#L136-L161

Description:

When a new management/performance fee or hurdle rate are set in the vault using **setFeeInfos**, the changes apply to the entire period since the last harvest. This means fees can be calculated using new values for time that has already passed, which users wouldn't expect.

For example, if 50 days go by after the last harvest and someone sets a new management fee that's twice as high, the next harvest will calculate fees using that higher rate for all 50 days. This could make users pay more than they should.

The same problem applies to the hurdle rate. If it was originally 6% (the vault needs to earn at least 6% before performance fees can be taken), but it's changed to 3%, the vault could take fees even if it didn't meet the original target.

```
    function setFeeInfos(address _feeRecipient, uint256 _managementFee, uint256
_performanceFee, uint256 _hurdleRate)
        external
        onlyOwner
    {
        require(_feeRecipient != address(0));
        require(_managementFee <= MAX_MANAGEMENT_FEE);
        require(_performanceFee <= MAX_PERFORMANCE_FEE);

        ManagedVaultStorage storage $ = _getManagedVaultStorage();
        if (feeRecipient() != _feeRecipient) {
            $.feeRecipient = _feeRecipient;
            emit FeeRecipientChanged(_msgSender(), _feeRecipient);
        }
        if (managementFee() != _managementFee) {
            $.managementFee = _managementFee;
            emit ManagementFeeChanged(_msgSender(), _managementFee);
        }
        if (performanceFee() != _performanceFee) {
            $.performanceFee = _performanceFee;
```

```
            emit PerformanceFeeChanged(_msgSender(), _performanceFee);
        }
        if (hurdleRate() != _hurdleRate) {
            $.hurdleRate = _hurdleRate;
            emit HurdleRateChanged(_msgSender(), _hurdleRate);
        }
    }
```

Remediation:

Ideally call **_harvestPerformanceFeeShares** before updating the new fee information during **setFeeInfos** to sync it to the current timestamp.

```
function setFeeInfos(address _feeRecipient, uint256 _managementFee, uint256
_performanceFee, uint256 _hurdleRate)
        external
    {
++      _harvestPerformanceFeeShares();
        require(_feeRecipient != address(0));
        require(_managementFee <= MAX_MANAGEMENT_FEE);
        require(_performanceFee <= MAX_PERFORMANCE_FEE);

        ManagedVaultStorage storage $ = _getManagedVaultStorage();
        if (feeRecipient() != _feeRecipient) {
            $.feeRecipient = _feeRecipient;
            emit FeeRecipientChanged(_msgSender(), _feeRecipient);
        }
        if (managementFee() != _managementFee) {
            $.managementFee = _managementFee;
            emit ManagementFeeChanged(_msgSender(), _managementFee);
        }
        if (performanceFee() != _performanceFee) {
            $.performanceFee = _performanceFee;
            emit PerformanceFeeChanged(_msgSender(), _performanceFee);
```

```
        }
        if (hurdleRate() != _hurdleRate) {
            $.hurdleRate = _hurdleRate;
            emit HurdleRateChanged(_msgSender(), _hurdleRate);
        }
    }
```

LOGLAB3-3 | Duplicate onlyOwner modifier in setAgent

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

src/VaultRegistry.sol:setAgent#L92-L97

Description:

In the **setAgent** function of **VaultRegistry**, the **onlyOwner** modifier is used twice.

```
function setAgent(address _agent) public onlyOwner noneZeroAddress(_agent)
onlyOwner {
    if (agent() != _agent) {
        _getVaultRegistryStorage().agent = _agent;
        emit AgentSet(_agent);
    }
}
```

Remediation:

Remove the second occurrence of onlyOwner:

```
-- function setAgent(address _agent) public onlyOwner noneZeroAddress(_agent)
onlyOwner {
++ function setAgent(address _agent) public noneZeroAddress(_agent) onlyOwner {
```


hexens
x
logarithm