Smart Contract Audit Report

Date of Audit: Oct 23, 2025
Version: v1.1
Audited by: Shahaf Antwarg, Liron Achdut
Client: textile, inc.
Repository: https://github.com/textile-protocol/textile-monorepo
Frozen Hash: 1bb98a06bdb5ebc9b2f1e786b9900c45cdc59469
Contracts Reviewed:
textile-protocol/textile-monorepo/tree/dev/packages/protocol/contracts/v2.1

Report Properties

Client	textile, inc.
Version	v2.1
Contracts	packages/protocol/contracts/v2.1
Author	Shahaf Antwarg, Liron Achdut
Auditors	Shahaf Antwarg, Liron Achdut
Classification	Confidential

Version Info

Version	Commit	Date	Author	Description
v1.0	1bb98a06bdb5ebc9b2f1e786b9900c45cdc59469	25.10.2025	bnolens	Release Candidate
v1.1	3847255235ea1ae8d6f73e7fb99612d4f8ce4c61	17.12.2025	bnolens	Final Release

Deployments

Contract	Chain	Address	Date

Table of Contents

1. Introduction

Following our request to review the source code of Textile's on-chain lending infrastructure, this report outlines our systematic approach to evaluating potential security issues, semantic inconsistencies between the design and implementation, and recommendations for improvements in both security and performance. Our review focused on the core contracts that handle deposits, loan issuance, interest accrual and collection, and proxy-based vault deployment. Overall, while the contracts exhibit many best practices - including role-based access control, UUPS upgrade safety, and non-reentrant transfer patterns - the audit identified critical vulnerabilities involving checkpoint updates, interest calculations, and governance risks that require architectural and code improvements before the protocol can be securely deployed.

1.1 About Textile

Textile is a decentralized lending protocol that transforms the traditional private credit market into an open, programmable capital graph. In Textile, every participant, whether an individual or institution, can become part of the credit supply chain without relying on traditional intermediaries.

Textile's mission is to decentralize trust and underwriting, allowing credit to reach the far "edges" of the economy where it's needed most, by lowering the technical barrier to participation so anyone can join.

1.2 About node.security

node.security Audits is a dedicated team specializing in smart contract security and blockchain risk management. Our team leverages a combination of automated tools and manual review to evaluate contracts thoroughly. We adhere to industry best practices and continuously update our methodologies based on emerging threats and vulnerabilities. For any queries or additional information, we can be reached via Telegram.

1.3 Disclaimer

This audit represents an independent security review of the smart contracts provided and is not a substitute for the complete functional testing that should be performed before any software release. Although every effort has been made to identify vulnerabilities, no audit can guarantee that all potential issues have been discovered. We strongly recommend additional independent audits and a public bug bounty program to further strengthen the security posture. This report is intended solely for security evaluation purposes and should not be interpreted as investment advice.

2. Report Structure

This audit report is organized to facilitate clear and efficient navigation from the most critical findings to less significant issues. Each issue is carefully documented with its severity rating and status, ensuring that readers can quickly assess the overall security posture and prioritize remediation efforts.

2.1 Issue tags

Issues are tagged as:

- **Resolved**: The issue has been fixed.
- **Unresolved**: The issue remains open.
- **Verified**: The functionality has been reviewed and confirmed with the client.

2.2 Severity levels

Severity levels are defined as follows:

- **Critical**: Issues that may lead to direct loss of funds or severe misallocation.
- **High**: Issues that significantly disrupt contract operation or pose a high risk of exploitation.
- **Medium**: Issues that affect the operation in non-catastrophic ways.
- **Low**: Minor issues that have minimal impact.
- **Info**: Observations that do not impact functionality but provide guidance for best practices.

2.3 Vulnerability Categories

Our audit followed a structured checklist covering a wide range of potential issues, from basic coding errors to advanced DeFi attack vectors. This approach ensures that every aspect of the code is thoroughly examined. The categories we evaluated include:

Basic Coding Bugs:
- Constructor and initializer mismatches
- Overflows, underflows, and unchecked arithmetic
- Short address or parameter attacks
- Uninitialized storage pointers

Semantic Consistency Checks:
- Consistency between the code and the design documentation
- Clarity of comments and correct parameter naming
- Detection of misleading or outdated code comments

Access Control & Authorization:
- Verification of role-based restrictions
- Proper management of privileged operations to prevent unauthorized access

Reentrancy & External Calls:
- Identification of reentrancy vulnerabilities
- Ensuring adherence to the checks-effects-interactions pattern

Business Logic & Financial Flow:
- Correct implementation of deposit, withdrawal, staking, and reward distribution
- Enforcement of lock periods, daily withdrawal caps, and minimum deposit amounts
- Ensuring consistency between internal state and actual token balances

Advanced DeFi Attack Vectors:
- Front-running, flash loan exploitation, and MEV risks
- Oracle manipulation and price feed vulnerabilities

Resource Management & Gas Optimization:
- Efficiency of loops and unbounded iterations
- Removal of redundant operations
- Appropriate use of compiler optimizations and internal function visibility

External Integration & Compatibility:
- Interaction with ERC20 tokens and potential issues with deflationary or rebasing tokens
- Proper handling of external contract calls and integration with critical services

Coding Standards & Documentation:
- Consistent naming conventions and parameter usage
- Correct and clear code comments without typos
- Adherence to best practices in coding style and organization

This comprehensive checklist enabled us to systematically identify and address vulnerabilities, ensuring a detailed and thorough audit of the contracts.

3. Scope of Work

Contracts Audited:

textile-protocol/textile-monorepo/tree/dev/packages/protocol/contracts/v2.1

Focus Areas:

1. **Access Control:** Admin/manager roles, upgrade permissions, vault-role assignments
2. **Interest Mechanics:** Compound-interest precision, accrual timing, rounding behavior
3. **Liquidity & Solvency:** Withdrawal limits, outstanding-debt checks, fund-reserve safeguards
4. **Module Integration:** Vault-collector interactions, permission scoping, multi-tenant isolation
5. **Initialization & Upgrades:** Proxy setup, initializer guards, UUPS authorization
6. **Token Interoperability:** Decimal normalization, allowance/transfer patterns, ERC-20 quirks
7. **Efficiency & Gas:** Loop bounds, redundant state, parameter validation

4. Methodology

1. **Manual Code Review**: Line-by-line inspection of each contract, focusing on fund flows (deposit, withdraw), lock enforcement, and external calls.
2. **Automated Analysis**: Tools such as Slither, Mythril for static analysis and vulnerability detection (reentrancy patterns, uninitialized storage, etc.).
3. **Threat Modeling**: Considering ways an attacker could bypass daily caps, lock periods, or manipulate callbacks.
4. **Testing & Simulation**: Deployed in a test environment, tried scenarios for normal usage and malicious attempts.
5. **Reporting**: Consolidation of all findings by severity, with recommended fixes and references to the code.

5. Findings

The audit uncovered several critical and high-severity vulnerabilities primarily related to missing checkpoint updates for non-compounding liquidity providers, incorrect interest calculations, and potential governance bypass mechanisms. These issues pose significant risks such as unauthorized fund withdrawals, balance manipulation, and disruption of protocol operations. The findings highlight the need for important architectural and code changes, as detailed in this report, to properly address these vulnerabilities and enhance the protocol's security and reliability. Addressing the items below will ensure the protocol is robust, transparent, and production-ready.

Below is a summary of the issues found. Each includes a severity rating (Critical, High, Medium, Low, Informational), title, category, explanation and our recommended remediation.

ID	Severity	Title	Category	Status
C-01	**Critical**	Missing Checkpoint Update for Non-Compounding LPs Minting	Business Logic	**Resolved**
C-02	**Critical**	Missing Checkpoint Update for Receiving Non-Compounding LPs	Business Logic	**Resolved**
C-03	**Critical**	Missing Burning of Shares on Auto-Claiming	Business Logic	**Resolved**
H-01	**High**	Claiming of Interest with 0 Shares Burnt	Business Logic	**Resolved**
H-02	**High**	Non-Compounding Interest Calculation Error	Financial Logic	**Resolved**
H-03	**High**	Inaccurate Binomial Interest Approximation	Financial Logic	**Resolved**
H-04	**High**	Micro-Loan Interest Rounding to Zero	Financial Logic	**Resolved**
M-01	**Medium**	Phantom Reserved Interest on Non-Compounding Withdrawal	Financial Logic	**Resolved**
M-02	**Medium**	Inactive Underwriter Causes Permanent DoS	Business Logic	**Resolved**
M-03	**Medium**	Missing Role Check for Protocol Admin Role Count Decrementing	Access Control	**Resolved**
M-04	**Medium**	Fee Bypass via Micro-Repayments	Financial Logic	**Resolved**
M-05	**Medium**	Phantom Reserved Shares from Request Update	Financial Logic	**Resolved**
M-06	**Medium**	Incorrect Withdrawal Limits	Financial Logic	**Resolved**
M-07	**Medium**	Pool Manager Can Frontrun Settings Approval	Access Control	**Resolved**
M-08	**Medium**	Emergency Withdrawal Timelock Risk	Business Logic	**Resolved**
M-09	**Medium**	Emergency Withdrawal Accounting Mismatch	Financial Logic	**Resolved**
M-10	**Medium**	Missing Pause Guards on Withdrawals	Financial Logic	**Resolved**

M-11	**Medium**	Double-Counting of Accumulated Rounding Dust	Financial Logic	**Resolved**
L-01	Low	Dust Threshold Calculation Error	Financial Logic	**Resolved**
L-02	Low	Flawed Dust Accumulation Mechanism	Financial Logic	**Resolved**
L-03	Low	maxWithdraw and maxRedeem Calculation Errors	Financial Logic	**Resolved**
L-04	Low	Last Admin Can Renounce Role	Access Control	**Resolved**
L-05	Low	Minimum Deposit Check Inconsistency	Business Logic	**Resolved**
L-06	Low	Deposits Allowed on Deactivated Tranche	Business Logic	**Resolved**
L-07	Low	Checks Covering for Calculation Mistakes	Code Quality	**Partially resolved**
L-08	Low	Missing Zero Interest Rate Validation	Financial Logic	**Resolved**
L-09	Low	Rounding Favoritism for Non-Compounding LPs	Financial Logic	**Resolved**
L-10	Low	Incorrect Utilization Rate Formula	Business Logic	**Resolved**
I-01	Info	Checks-Effects-Interactions Pattern Not Followed	Code Quality	**Resolved**
I-02	Info	Stale Pending Proposal Issue	Business Logic	**Unresolved**
I-03	Info	Inactive Tranche Not Removed from List	Code Quality	**Resolved**

C-01: Missing Checkpoint Update for Non-Compounding LPs Minting

Severity: Critical
Category: Business Logic

Description:
When a non-compounding LP that has existing shares mints/deposits, they get new shares but their checkpoint isn't updated. Which means they can claim interest for those new shares that haven't accumulated interest yet, thus effectively stealing that interest from the rest of the LPs by reducing share price. This is extremely dangerous because it will cause innocent users to steal funds without realizing, by depositing more funds and then claiming interest. Also the amount of interest stolen is proportional to the amount deposited, which is unlimited, so a malicious user can steal all the reserved interest in the tranche. When minting new shares, must update the minter checkpoint to a weighted average with current `reservedInterestPerShareAccumulated` to prevent this. Recommended to add weighting average in `_update` for the case of `toNonCompounding` and `!fromNonCompounding` (`fromNonCompounding` will be false in case `from` is zero address).

Example:

reservedInterestPerShareAccumulated = 10e18
User (non-compounding): 100 shares, checkpoint = 10e18
User's claimable: (10e18 - 10e18) × 100 / 1e18 = 0

New interest accrues:
reservedInterestPerShareAccumulated = 11e18
User's claimable: (11e18 - 10e18) × 100 / 1e18 = 100

User deposits 100 new shares.
After deposit:
User: 200 shares, checkpoint = 10e18 (unchanged)
User's claimable: (11e18 - 10e18) × 200 / 1e18 = 200
User gained extra 100 interest that the new shares never earned (or gain any extra X assets on any deposit of X new shares)

If User's checkpoint had been updated it would have been (10e18 × 100 + 11e18 × 100) / (100 + 100) = 10.5e18, resulting in claimable being (11e18 - 10.5e18) × 200 / 1e18 = 100 as expected

C-02: Missing Checkpoint Update for Receiving Non-Compounding LPs

Severity: Critical
Category: Business Logic

Description:
When a non compounding LP that has existing shares receives shares from a compounding LP, the receiver's checkpoint is not updated, resulting in the receiver being able to claim more interest than actually deserved, thus effectively stealing that interest from the rest of the LPs by reducing share price. Any transfer from a compounding LP to a non-compounding LP will result in the non-compounding LP receiving more interest than actually earned when claiming, even innocent users. Receiver's checkpoint should be weight averaged with `reservedInterestPerShareAccumulated` to prevent this.

Example:

User A (compounding): 100 shares
User B (non-compounding): 100 shares, checkpoint = 3e18
User B's claimable: (10e18 - 3e18) × 100 / 1e18 = 700
User A transfers 50 shares to User B.

After transfer:
User B: 150 shares, checkpoint = 3e18 (unchanged)
User B's claimable: (10e18 - 3e18) × 150 / 1e18 = 1,050
User B gained extra 350 interest that was never reserved

If User B's checkpoint had been updated it would have been (3e18 × 100 + 10e18 × 50) / (100 + 50) = 5333333333333333333, resulting in claimable being (10e18 - 5.333e18) × 150 / 1e18 = 700 as expected

C-03: Missing Burning of Shares on Auto-Claiming

Severity: Critical
Category: Business Logic

Description:
When a non-compounding LP sends shares to a compounding LP, there's an auto claim of the accumulated interest to the sender, but the shares proportional to the interest received are not burnt (as they are on a normal claim), allowing users to send shares to another compounding address they own and receiving assets without burning the proportional shares, thus effectively stealing that interest from the rest of the LPs by reducing share price. Add appropriate share burning to `_autoClaimOnTransfer`. Recommended to use an internal shared function that both `_autoClaimOnTransfer` and `_claimReservedInterestInternal` use, and call it before calling `_update`.

Example:

reservedInterestPerShareAccumulated = 10e18
totalAssets = 30,000
totalSupply = 300 shares
Share price = 30,000 / 300 = 100 assets/share
User A (non-compounding): 100 shares, checkpoint = 3e18
User B (compounding, controlled by A): 100 shares = 100 × 100 = 10,000 assets
User C (compounding, innocent LP): 100 shares = 100 × 100 = 10,000 assets
User A's claimable: (10e18 - 3e18) × 100 / 1e18 = 700 assets
User A transfers 100 shares to User B.

After transfer (with bug - no shares burned):
TotalSupply: 300 (unchanged)
totalAssets: 30,000 - 700 = 29,300
Share price: 29,300 / 300 = 97.67 assets/share (reduced)
User A: 0 shares, 700 assets claimed
User B: 200 shares = 200 × 97.67 = 19,534 assets
User C: 100 shares = 100 × 97.67 = 9,767 assets
User A+B combined: 700 + 19,534 = 20,234 assets
User C lost 233 from original 10,000

If shares had been burned correctly (like in normal claim): `sharesToBurn = convertToShares(700) = 700 / 100` = 7 shares
After transfer (with fix - shares burned):
totalSupply: 300 - 7 = 293 shares
totalAssets: 29,300
Share price: 29,300 / 293 = 100 assets/share (maintained)
User A: 0 shares, 700 assets claimed
User B: 200 - 7 = 193 shares = 193 × 100 = 19,300 assets
User C: 100 shares = 100 × 100 = 10,000 assets (no loss)

Also, since the sender's checkpoint is not updated, as stated in C-01, user A can send to user B most of his shares and keep 1 share, and then send back the shares from user B to user A and repeat the process until the entire reserved interest is drained.

H-01: Claiming of Interest with 0 Shares Burnt

Severity: High
Category: Business Logic

Description:
In `_claimReservedInterestInternal`, revert in case `sharesToBurn == 0`. Otherwise a fee will be transferred to the user without burning any shares, in case the claimable amount is smaller than the share price. This is possible if the number of user shares multiplied by the accumulated interest per share since last checkpoint is smaller than the share price. This allows a user to claim his percentage of the reserved interest without burning any shares, thus effectively stealing that interest from the rest of the LPs by reducing share price. This means the user can redeem those shares and earn double on them.

Example:
totalAssets = 100,000
totalSupply = 1000 shares
Share price = 100,000 / 1000 = 100 assets/share
totalSharesWithDisabledCompounding = 1000 (100% of all shares)

User keeps 200 accounts with 1 share each (20% of all shares with disabled compounding) and waits for an interest accrual smaller than share price (claims interest as usual until that happens), then claims interest on all accounts at once and receives it without burning any shares, receiving his percent of the non-compounding shares of the newly accrued interest:

An interest of 10000 assets is paid which 10000 of becomes reserved for non compounding, and yields 10000/1000 = 10 assets reserved per share

totalAssets = 110,000
totalSupply = 1000 shares
Share price = 110,000 / 1000 = 110 assets/share
totalSharesWithDisabledCompounding = 1000 (100% of all shares)

Each claim per 1 share has only latest checkpoint and so it's claimable amount is 10 * 1 = 10 assets, and since `sharesToBurn = convertToShares(claimable)` it would burn 0 assets. Resulting in the user claiming 200 * 10 = 2000 assets without burning any shares, and reducing share price. After the claims:

totalAssets = 108,000
totalSupply = 1000 shares
Share price = 108,000 / 1000 = 108 assets/share
totalSharesWithDisabledCompounding = 1000 (100% of all shares)

If the user had one account with 100 shares, this interest claim would result in claimable being the same 10 * 200 = 2000 assets and 2000 / 110 = 18 shares burnt. This means the user would have been left with 182 shares after the claim. But the user got to keep these 18 shares and can now redeem them, earning an extra 18 * 108 = 1944 assets. In this scenario, after the claim:

totalAssets = 108,000
totalSupply = 982 shares
Share price = 108,000 / 982 = 109.979633401 assets/share (slightly reduced due to floor rounding)

H-02: Non-Compounding Interest Calculation Error

Severity: High
Category: Financial Logic

Description:
Interest is not really compounding but only compounding per each accrual period, only on the outstanding debt (principal, not including accumulated interest). So the more frequent the accruals, the smaller the interest will be since it's only compounded over the period since the last accrual. This allows the borrower to frequently call `accrueInterest` and reduce the interest they owe. To have true compounding, the `_acumulatedInterest` must be added into the calculation of the new interest.

Example:

Interest rate is set to 20% annually
Borrower does a drawdown of 10,000$
Borrower calls `accrueInterest` once a month for a year:

Month 1: newInterest = 10,000 - 10,000*(1 + 20%/seconds per year)^(seconds in 1 month) = 168.17589552
And since the outstandingDebt hasn't changed, for every month following, the calculation would be the same:
Month 2: newInterest = 10,000 - 10,000*(1 + 20%/seconds per year)^(seconds in 1 month) = 168.17589552
And so on, resulting in the interest accumulated over the year to be 168.17589552 * 12 = 2,018.1107462

Whereas if the borrower had called `accrueInterest` once after a year that would result in the interest being:
newInterest = 10,000*(1 + 20%/seconds per year)^(seconds in 1 year) - 10,000 = 2,214.02757386
Resulting in a loss of 195.91682766 of interest that should have been paid.

If the calculation had included the interest in the outstanding debt, that would result in the following interests accrued:

Month 1: newInterest = 10,000 - 10,000*(1 + 20%/seconds per year)^(seconds in 1 month) = 168.17589552
Month 2: newInterest = 10,168.17589552 × (1 + 20%/seconds per year)^(seconds in 1 month) - 10,168.17589552 = 171.02358968
Month 3: newInterest = 10,339.19948520 × (1 + 20%/seconds per year)^(seconds in 1 month) - 10,339.19948520 = 173.88118210
Month 4: newInterest = 10,513.08066730 × (1 + 20%/seconds per year)^(seconds in 1 month) - 10,513.08066730 = 176.76904603
Month 5: newInterest = 10,689.84971333 × (1 + 20%/seconds per year)^(seconds in 1 month) - 10,689.84971333 = 179.68222889
Month 6: newInterest = 10,869.53194222 × (1 + 20%/seconds per year)^(seconds in 1

month) - 10,869.53194222 = 182.62399555
Month 7: newInterest = 11,052.15593777 × (1 + 20%/seconds per year)^(seconds in 1 month) - 11,052.15593777 = 185.59473467
Month 8: newInterest = 11,237.75067244 × (1 + 20%/seconds per year)^(seconds in 1 month) - 11,237.75067244 = 188.59596551
Month 9: newInterest = 11,426.34663795 × (1 + 20%/seconds per year)^(seconds in 1 month) - 11,426.34663795 = 191.62807567
Month 10: newInterest = 11,617.97471362 × (1 + 20%/seconds per year)^(seconds in 1 month) - 11,617.97471362 = 194.69250958
Month 11: newInterest = 11,812.66722320 × (1 + 20%/seconds per year)^(seconds in 1 month) - 11,812.66722320 = 197.79072702
Month 12: newInterest = 12,010.45795022 × (1 + 20%/seconds per year)^(seconds in 1 month) - 12,010.45795022 = 200.92320008

Total interest accrued over the year: 168.17589552 + 171.02358968 + 173.88118210 + 176.76904603 + 179.68222889 + 182.62399555 + 185.59473467 + 188.59596551 + 191.62807567 + 194.69250958 + 197.79072702 + 200.92320008 = 2,211.38115030

The correct monthly compounding yields 2,211.38115030, which is very close to the yearly accrual (2,214.02757386). The small difference is due to the difference between discrete monthly periods and 1 year.

H-03: Inaccurate Binomial Interest Approximation

Severity: High
Category: Financial Logic

Description:
Add the fourth order to the binomial approximation or use direct calculation without approximation for interest accuracy. For large APRs and/or large principal amounts the difference is substantial. See following examples:

$10,000 PRINCIPAL

TABLE 1A.1: Interest Earned - Binomial Approximation

Rate	1 Day	1 Week	1 Month	3 Months	6 Months	1 Year	2 Years	5 Years
1% APR	$0.27	$1.92	$8.22	$24.69	$49.44	$100.50	$202.00	$512.50
5% APR	$1.37	$9.59	$41.18	$124.05	$249.63	$512.66	$1,051.25	$2,832.10
10% APR	$2.74	$19.20	$82.53	$249.64	$505.50	$1,051.62	$2,212.96	$6,452.55
20% APR	$5.48	$38.43	$165.74	$505.51	$1,036.54	$2,213.33	$4,906.63	$16,666.17
50% APR	$13.71	$96.35	$419.52	$1,312.00	$2,794.73	$6,458.30	$16,666.43	$82,287.96
100% APR	$27.43	$193.63	$856.62	$2,794.74	$6,347.38	$16,666.64	$53,333.11	$383,329.79

TABLE 1A.2: Interest Earned - Exact Power

Rate	1 Day	1 Week	1 Month	3 Months	6 Months	1 Year	2 Years	5 Years
1% APR	$0.27	$1.92	$8.22	$24.69	$49.44	$100.50	$202.01	$512.71
5% APR	$1.37	$9.59	$41.18	$124.05	$249.64	$512.71	$1,051.71	$2,840.25
10% APR	$2.74	$19.20	$82.53	$249.64	$505.51	$1,051.71	$2,214.03	$6,487.21
20% APR	$5.48	$38.43	$165.74	$505.51	$1,036.58	$2,214.03	$4,918.25	$17,182.82
50% APR	$13.71	$96.35	$419.52	$1,312.10	$2,796.36	$6,487.21	$17,182.82	$111,824.94
100% APR	$27.43	$193.63	$856.64	$2,796.36	$6,374.67	$17,182.82	$63,890.56	$1,474,131.47

TABLE 1A.3: Interest Loss (Exact - Binomial)

Rate	1 Day	1 Week	1 Month	3 Months	6 Months	1 Year	2 Years	5 Years
1% APR	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.01	$0.21
5% APR	$0.00	$0.00	$0.00	$0.00	$0.01	$0.05	$0.45	$8.15
10% APR	$0.00	$0.00	$0.00	$0.00	$0.01	$0.09	$1.06	$34.66
20% APR	$0.00	$0.00	$0.00	$0.00	$0.04	$0.70	$11.61	$516.65
50% APR	$0.00	$0.00	$0.00	$0.10	$1.62	$28.91	$516.39	$29,536.98

100% APR	$0.00	$0.00	$0.02	$1.62	$27.29	$516.18	$10,557.45	$1,090,801.68

$100,000 PRINCIPAL

TABLE 1B.1: Interest Earned - Binomial Approximation

Rate	1 Day	1 Week	1 Month	3 Months	6 Months	1 Year	2 Years	5 Years
1% APR	$2.74	$19.18	$82.23	$246.88	$494.37	$1,005.00	$2,020.00	$5,125.00
5% APR	$13.70	$95.94	$411.80	$1,240.50	$2,496.34	$5,126.57	$10,512.55	$28,321.02
10% APR	$27.40	$191.96	$825.30	$2,496.40	$5,055.05	$10,516.20	$22,129.63	$64,525.54
20% APR	$54.81	$384.30	$1,657.42	$5,055.10	$10,365.40	$22,133.29	$49,066.35	$166,661.68
50% APR	$137.08	$963.52	$4,195.19	$13,119.99	$27,947.33	$64,583.04	$166,664.29	$822,879.57
100% APR	$274.35	$1,936.32	$8,566.21	$27,947.36	$63,473.80	$166,666.38	$533,331.06	$3,833,297.89

TABLE 1B.2: Interest Earned - Exact Power

Rate	1 Day	1 Week	1 Month	3 Months	6 Months	1 Year	2 Years	5 Years
1% APR	$2.74	$19.18	$82.23	$246.88	$494.37	$1,005.02	$2,020.13	$5,127.11
5% APR	$13.70	$95.94	$411.80	$1,240.51	$2,496.40	$5,127.11	$10,517.09	$28,402.54
10% APR	$27.40	$191.96	$825.30	$2,496.40	$5,055.13	$10,517.09	$22,140.28	$64,872.13
20% APR	$54.81	$384.30	$1,657.42	$5,055.13	$10,365.80	$22,140.28	$49,182.47	$171,828.18
50% APR	$137.08	$963.52	$4,195.20	$13,120.98	$27,963.56	$64,872.13	$171,828.18	$1,118,249.37
100% APR	$274.35	$1,936.32	$8,566.40	$27,963.56	$63,746.72	$171,828.18	$638,905.59	$14,741,314.73

TABLE 1B.3: Interest Loss (Exact - Binomial)

Rate	1 Day	1 Week	1 Month	3 Months	6 Months	1 Year	2 Years	5 Years
1% APR	$0.00	$0.00	$0.00	$0.00	$0.00	$0.02	$0.13	$2.11
5% APR	$0.00	$0.00	$0.00	$0.01	$0.06	$0.54	$4.55	$81.52
10% APR	$0.00	$0.00	$0.00	$0.01	$0.08	$0.89	$10.64	$346.59
20% APR	$0.00	$0.00	$0.00	$0.03	$0.41	$6.98	$116.12	$5,166.50
50% APR	$0.00	$0.00	$0.01	$0.99	$16.23	$289.09	$5,163.89	$295,369.81
100% APR	$0.00	$0.00	$0.19	$16.20	$272.93	$5,161.80	$105,574.53	$10,908,016.84

$1,000,000 PRINCIPAL

TABLE 1C.1: Interest Earned - Binomial Approximation

Rate	1 Day	1 Week	1 Month	3 Months	6 Months	1 Year	2 Years	5 Years

1% APR	$27	$192	$822	$2,469	$4,944	$10,050	$20,200	$51,250
5% APR	$137	$959	$4,118	$12,405	$24,963	$51,266	$105,125	$283,210
10% APR	$274	$1,920	$8,253	$24,964	$50,550	$105,162	$221,296	$645,255
20% APR	$548	$3,843	$16,574	$50,551	$103,654	$221,333	$490,663	$1,666,617
50% APR	$1,371	$9,635	$41,952	$131,200	$279,473	$645,830	$1,666,643	$8,228,796
100% APR	$2,743	$19,363	$85,662	$279,474	$634,738	$1,666,664	$5,333,311	$38,332,979

TABLE 1C.2: Interest Earned - Exact Power

Rate	1 Day	1 Week	1 Month	3 Months	6 Months	1 Year	2 Years	5 Years
1% APR	$27	$192	$822	$2,469	$4,944	$10,050	$20,201	$51,271
5% APR	$137	$959	$4,118	$12,405	$24,964	$51,271	$105,171	$284,025
10% APR	$274	$1,920	$8,253	$24,964	$50,551	$105,171	$221,403	$648,721
20% APR	$548	$3,843	$16,574	$50,551	$103,658	$221,403	$491,825	$1,718,282
50% APR	$1,371	$9,635	$41,952	$131,210	$279,636	$648,721	$1,718,282	$11,182,494
100% APR	$2,743	$19,363	$85,664	$279,636	$637,467	$1,718,282	$6,389,056	$147,413,147

TABLE 1C.3: Interest Loss (Exact - Binomial)

Rate	1 Day	1 Week	1 Month	3 Months	6 Months	1 Year	2 Years	5 Years
1% APR	$0	$0	$0	$0	$0	$0	$1	$21
5% APR	$0	$0	$0	$0	$1	$5	$45	$815
10% APR	$0	$0	$0	$0	$1	$9	$106	$3,466
20% APR	$0	$0	$0	$0	$4	$70	$1,161	$51,665
50% APR	$0	$0	$0	$10	$162	$2,891	$51,639	$2,953,698
100% APR	$0	$0	$2	$162	$2,729	$51,618	$1,055,745	$109,080,168

H-04: Micro-Loan Interest Rounding to 0

Severity: High
Category: Financial Logic

Description:
The interest calculation converts the compounded amount from RAY precision (1e27) to standard precision through integer division, which rounds down to zero for small principal amounts over short time periods. For example, a loan of $13.14 USDC (for 6-decimal tokens) will accrue 0 interest if `accrueInterest()` is called every 12 seconds, while a $2.63 USDC loan accrues 0 with per-minute calls. Borrowers can exploit this by either taking micro-loans below these thresholds or by frequently calling `accrueInterest()` on larger loans to prevent any interest from accumulating. Consider implementing a minimum loan amount to prevent micro-loans, and/or adding a minimum accrual interval to prevent griefing through frequent zero-interest accruals.

M-01: Phantom Reserved Interest on Non-Compounding Withdrawal

Severity: Medium
Category: Financial Logic

Description:
When a non-compounding LP withdraws/redeems, the amount that was reserved per the redeemed shares doesn't get subtracted from `reservedForNonCompoundingInterest`, and so remains reserved so that the borrower can't borrow it, and other LPs can't withdraw it. The reserved amount can't be unreserved in any way and so remains stuck in the contract, since the reserved amount can't be reduced. This means that the only way to get the last funds in the contract below this amount is to do an emergency withdrawal, but even this doesn't fix the accounting—the phantom reservation in `reservedForNonCompoundingInterest` persists permanently, making the contract accounting permanently broken.

M-02: Inactive Underwriter Causes Permanent DoS

Severity: Medium
Category: Business Logic

Description:
If an underwriter of a pool makes himself inactive (by calling `UnderwriterRegistry.deactivate`) then the pool's functionality will be permanently disabled since `drawdown`, `repay`, `repayInterest`, `repayAll` will all revert on `getUnderwriterFees`, and there's no mechanism to revoke the underwriter once set (also depositing/minting to the tranche will revert). Add a check that underwriter is active to `calculateInterestFees` (and not only that it is set) and consider adding a pool manager function to revoke an inactive underwriter so that a new one can be set. Also consider allowing the manager to always be able to revoke the underwriter

M-03: Missing Role Check for Protocol Admin Role Count Decrementing

Severity: Medium
Category: Access Control

Description:
In `renounceRole`, the `_protocolAdminCount` is decremented whether the account had the role or not. This way this function can be called with any sender address and still decrement the `_protocolAdminCount` until it is equal to 1, which will not allow anyone else to renounce the protocol admin role. This is also irreversible and the count can't be recovered to the right count.

M-04: Fee Bypass via Micro-Repayments

Severity: Medium
Category: Financial Logic

Description:
The borrower (or anyone) can repay the interest in tiny amounts, resulting in protocol and underwriter fees being less than 1 wei and getting rounded down to 0. This way the whole mechanism of fees can be overridden and fees never paid, and the total interest paid will be less than it should have been since no fees will be deducted and it will all go to net interest. This allows the borrower to practically steal the fees from the protocol. Recommended to add a minimum amount to repayments, which should include the remainder of the debt, meaning: revert if `amount < MIN_REPAYMENT && amount < totalOwed` on `repay` (to alow a payment of less then the minimum in case the rest of the debt is less then the minimum) and revert if `interestOwed < MIN_REPAYMENT` on `repayInterest`. There's no need for a limit on `repayAll` since it clears the debt and no new interest will be accumulated until next drawdown, and also it should have no limit since it's meant for borrowers to close their position. For the smallest fee possible, of 1/10000, a minimum of 10000 wei is enough to prevent this from happening and fees to be paid. Note that fees are still rounded down, so there could still be a loss of 1 wei per repayment.

M-05: Phantom Reserved Shares from Request Update

Severity: Medium
Category: Financial Logic

Description:
If a user had a withdrawal request approved, but then instead of withdrawing made a new withdrawal request, the request is updated with the new requested amount and the approval status is set to false, but `totalReservedShares[tranche]` is not updated. This leaves phantom shares reserved that are locked away and not reserved for anyone, and can't be borrowed or withdrawn. This permanent accounting corruption accumulates over time with no recovery mechanism To fix: in `createRequest`, check if the user had a previously approved request and if so, deduct the amount from `totalReservedShares[tranche]`.

M-06: Incorrect Withdrawal Limits

Severity: Medium
Category: Financial Logic

Description:
The amount of assets that is withdrawn/redeemed is only checked to be smaller than the virtual balance, and not smaller than `getAvailableForBorrowing()` as is checked in `approveRedeemRequest`. This can result in users withdrawing assets that have been previously reserved for non-compounding interest. Fix by checking withdrawal/redeem amounts are smaller than (fixed) `maxWithdraw` or `maxRedeem`.

M-07: Pool Manager Can Frontrun Settings Approval

Severity: Medium
Category: Access Control

Description:
Pool manager can propose tranche or credit line settings change for underwriter's approval, and frontrun the underwriter's approval call with a proposal for different settings, and so get any settings approved. Consider including the settings or a settings id/nonce in the approve function.

M-08: Emergency Withdrawal Timelock Risk

Severity: Medium
Category: Business Logic

Description:
Remove timelock on reserve's emergency withdrawal (and so get rid of the `EmergencyWithdrawal` struct and `EMERGENCY_TIMELOCK` constant). In case of emergency where the guardian wants to withdraw funds from the reserve, you don't want to have to wait 3 days before the funds could be retrieved. Also, In case the reserve balance has decreased in those 3 days below the requested amount, the emergency withdrawal would fail and a new request will have to be submitted.

M-09: Emergency Withdrawal Accounting Mismatch

Severity: Medium
Category: Financial Logic

Description:
Emergency withdrawal from the reserve does not update the tranche's virtual balance and interest reserved for non-compounding, resulting in an accounting mismatch. It will leave a false share price and a bank run scenario - only the first users to withdraw will succeed until reserve is empty and then any withdrawals, claiming of reserved interest or drawdowns will fail. On emergency withdrawal, consider updating the tranche's virtual balance and interest reserved for non-compounding, and adding a factor to reduce by when calculating `claimable`. It could be left as is and pause all functionality until the funds are returned to the reserve and normal functionality is restored. Consider which scenarios the emergency withdrawal is meant for and how you want to deal with them.

M-10: Missing Pause Guards on Withdrawals

Severity: Medium
Category: Financial Logic

Description:
`withdraw`, `redeem` and `claimReservedInterest` (and also transfers) in tranche are not marked `whenPoolNotPaused` and `whenNotShutdown`. Consider whether you want to allow users to withdraw during pause/shutdown, especially since emergency withdrawal creates a bank run situation in its current implementation. Note that withdrawal requests can be paused at the protocol level at the withdrawal registry.

M-11: Double-Counting of Accumulated Rounding Dust

Severity: Medium
Category: Financial Logic

Description:
When `_accumulatedRoundingDust` passes the threshold, it is added to `virtualBalance`, but that amount has already been added to the virtual balance as part of the net interest in `recordInterest`. This causes this dust to be counted twice in `virtualBalance` and so inflating `totalAssets' and raising the share price without having the assets in the reserve to back it up, resulting in withdrawal failures. Remove this line.

L-01: Dust Threshold Calculation Error

Severity: Low
Category: Financial Logic

Description:
Dust threshold is supposed to be `1 unit of the underlying asset (e.g., 1 USDC = 1e6)` as the documentation states, but it's calculated as `dustThreshold = 10**decimals()`, where decimals are calculated as `_underlyingDecimals + _decimalsOffset()`. This will result as the dust threshold being 10**(6+7) in the case of USDC as the underlying asset, so 10**7 = 10,000,000 USDC. To fix, set the dust threshold as `10 ** _underlyingDecimals`, preferably as an immutable state variable set on initialization and not calculated every time.

L-02: Flawed Dust Accumulation Mechanism

Severity: Low
Category: Financial Logic

Description:
The whole mechanism of dust accumulation is flawed, it's not possible to redistribute that dust back to compounding LPs since it has been reserved for non-compounding LPs, and could have been all claimed. Remove this mechanism and choose whether you want to round up or down the `reservedPerShare.` The difference can be 1 wei at most for every interest repayment and is negligible and doesn't create a real unfair advantage for non-compounding LPs.

L-03: maxWithdraw and maxRedeem Calculation Errors

Severity: Low
Category: Financial Logic

Description:
There are 3 mistakes in calculation of `maxWithdraw` and `maxRedeem`:

1. You need to use `getAvailableForBorrowing()` and not `virtualBalance`, and include the user's approved assets, as is in `approveRedeemRequest`
2. In the case of no approval needed there is a relative amount calculated that is wrong and not what is actually available
3. In the case that approval is required there should also be consideration for the user's current balance, since that may change since the request has been done.

To fix:

In `maxWithdraw`, in case of `!_settings.requireApprovalForWithdrawals`, the maximal withdrawal amount should be the actual allowed by `withdraw`, which is `Math.min(getAvailableForBorrowing(), ownerAssets)`

or `Math.min(getAvailableForBorrowing() + approvedAssets, Math.min(approvedAssets, ownerAssets)` in case of approval needed.

Same for `maxRedeem`, should be the actual allowed by `redeem`, which is `Math.min(_convertToShares(getAvailableForBorrowing()), balanceOf(owner)`

or `Math.min(_convertToShares(getAvailableForBorrowing()) + request.shares, Math.min(request.shares, balanceOf(owner))` in case of approval needed.

L-04: Last Admin Can Renounce Role

Severity: Low
Category: Access Control

Description:
There's no check in `revokeRole` that the account is not the last admin. This can allow for the only admin to renounce his role, thinking it will revert if he's the last, but actually it will leave the protocol without an admin. Recommended to only override `_revokeRole` that's called by `revokeRole` and `renounceRole` and have the check there.

L-05: Minimum Deposit Check Inconsistency

Severity: Low
Category: Business Logic

Description:
`deposit` checks that gross assets are less than minimum deposit and `mint` checks that net assets are less than minimum deposit. Both should check the same (to not allow users to deposit less than minimal amount with `deposit` in case of wanting minimal net amount, or blocking users from depositing the minimal amount with `mint` in case of wanting minimal gross amount). Recommended to use gross amounts for better UX.

L-06: Deposits Allowed on Deactivated Tranche

Severity: Low
Category: Business Logic

Description:
Deposits/mints are still allowed on a deactivated tranche. A tranche can only be deactivated when it's empty and there are no deployed funds, meaning `vitrualBalance` != 0, `virtualDeployed` != 0, but after deactivation it is still possible to call `deposit` or `mint` and increase the tranche's `vitrualBalance`. Consider adding a check that the tranche is active to `deposit` and `mint`. It is also possible to leave it as is, since users can withdraw the funds they deposited to a deactivated tranche, and no interest will be accrued since there are no deployed funds and no possibility to drawdown when the tranche is deactivated.

L-07: Checks Covering for Calculation Mistakes

Severity: Low
Category: Code Quality

Description:
Most uses of `Math.min` are redundant and are contradicting assumptions, they will cover up in case of calculation mistakes and should be removed.

- `Math.min(virtualBalance, _reserve.balance());` should be replaced with `virtualBalance` since `virtualBalance` should always reflect the `_reserve.balance()`, unless there's been an emergency withdrawal.
- `Math.min(request.shares/oldShares/sharesConsumed, totalReservedShares[tranche])` should be replaced with `request.shares/oldShares/sharesConsumed`
- `Math.min(shares, request.shares)` should be replaced with `shares`
- `Math.min(value, totalSharesWithDisabledCompounding)` should be replaced with `value`

There are also many cases of checks in the style of `A > B ? A - B : 0` where the case that `A < B` should not be possible, so this check should not cover for the impossible case (and should be replaced with simply `A - B`, or the expected value).
For example:

- `if (sharesToBurn > userShares) sharesToBurn = userShares;` this should not be possible since `convertToAssets(userShares - sharesToBurn)` should be the user's initial investment. You should have a test that checks this assumption and not a cover up in case it goes wrong, since it will allow the user to receive more assets then shares burnt and decrease share price.
- `totalDeployed = totalDeployed >= principalPaid ? totalDeployed - principalPaid : 0` just use `totalDeployed -= principalPaid` since totalDeployed < principalPaid is not possible. If it does happen an underflow should catch this instead of covering up.
- `if (reservedInterestPerShareAccumulated <= checkpoint) return;` should be changed to `if (reservedInterestPerShareAccumulated == checkpoint) return;` since `reservedInterestPerShareAccumulated` cannot decrease.

L-08: Missing Zero Interest Rate Validation

Severity: Low
Category: Financial Logic

Description:
Low: `validateCreditLineSettings` should check that interest rate per second is not zero (`(interestRate * RAY / BPS_DIVISOR) / SECONDS_PER_YEAR != 0`) and not only that interest rate is not zero. Otherwise `compoundPerSecondRayBinomial` always returns the outstanding debt amount without addition, resulting in zero interest.

L-09: Rounding Favoritism for Non-Compounding LPs

Severity: Low
Category: Financial Logic

Description:
When interest is repaid and `reservedForNonCompounding` is calculated, if less than 1 wei should be reserved for non-compounding still 1 wei is reserved since the calculation is rounded up. For example if 1 wei is repaid, and there are 99 compounding shares and 1 non-compounding, still the full 1 wei is reserved for non-compounding, even though only 0.01 should have been essentially reserved. This creates a tiny disadvantage in favor of non-compounding LPs, which is negligible even after many attempts (1M repayments of 1 wei net each can yield an advantage of $1 at most for all non-compounding LPs: $0.99999 if non-compounding are 0.001% of all shares). Also since the accumulated amount is reserved for non-compounding, it is reduced from the total available for borrowing. Consider adding a multiplier for `reservedForNonCompounding`, same as done for `reservedInterestPerShareAccumulated`

L-10: Incorrect Utilization Rate Formula

Severity: Low
Category: Business Logic

Description:
`utilizationRate` calculation in `getPoolMetrics` is wrong - should be divided by `drawLimit` and not `totalAssets` , as in `getUtilizationRate()`)

I-01: Checks-Effects-Interactions Pattern Not Followed

Severity: Info
Category: Code Quality

Description:
It's recommended to use CEI pattern in `drawdown`, `repay`, `repayInterest`, `repayAll` as it is best practice.

I-02: Stale Pending Proposal Issue

Severity: Info
Category: Business Logic

Description:
Managers can propose a creditLine/Tranche setting that requires approval, and then propose another one that gets immediately set without requiring approval, but the previous one will still be pending and can be approved by the underwriter even if it's not relevant anymore. If this is not desired behavior, reject the previous pending proposal when approving a new setting.

I-03: Inactive Tranche Not Removed from List

Severity: Info
Category: Code Quality

Description:
`deactivateTranche` does not remove tranche from `trancheList` so list size stays the same, which makes it impossible to add another tranche later (and reactivation is not supported) in a single tranche implementation, and would make `_getActiveTranches` less efficient when multi tranche is supported.

6. Additions

Structure recommendations

These changes are recommended to simplify and improve code structure:

1. Use openZeppelin's `TimelockController` instead of `UpgradeTimelock` & `UpgradeableBeaconWithTimelock`
2. Join `ProtocolConfiguration` under `Registry` (to not have `emergencyShutdown` in a separate, not mandatory contract)
3. Consider joining `WithdrawalRegistry` into `Tranche` for efficiency (no need for inter-contract calls)
4. Consider following the YAGNI principle and removing any view functions that aren't in use by the ui atm.
5. The multi-tranche implementation is nowhere near ready, and making the current single-tranche version cumbersome, inefficient and more prone to bugs. Consider whether you want to keep it or remove everything regarding it for the current version and create a new multi-tranche version later on.

Code improvements

The following changes are recommended to improve code quality, making code more functional, efficient, readable, removing code duplication or improving UX:

1. Remove unused or unnecessary imports, e.g. in `StructuredPool` importing `AccessControlUpgradeable` which is already imported by `AccessControlHelpersUpgradeable`, and `Reserve` which is not used. **Only examples provided removed**
2. Use `InterestLogic` functions without `InterestState` and instead use the necessary vars directly, and update `_lastInterestAccrualTime` in `StructuredPool`
3. override `_grantRole` and not `grantRole` - **Resolved**
4. `revokeRole` can only be called by role admins of protocol admin role, which means only protocol admins can call it and so you don't need to check whether the caller had the role. Same for `grantRole`.
5. remove all basis points constants and hardcoded 10000 and use `MathUtils.BPS_DIVISOR` - **Resolved**
6. In 'grantRole' there's no update to `_lastAddedProtocolAdmin`, this is currently keeping the first added protocol admin, if this is desired then change the var name to be `_firstAddedProtocolAdmin` - **Resolved**
7. Keep one `RAY` constant in `MathUtils` - **Resolved**
8. Remove `withdrawalQueue` and `nextWithdrawalId` that are never used - **Resolved**

9. Use openZeppelin's `AccessControlEnumerable` instead of manually counting in `AccessControlHelpers`
10. Use openZeppelin's `_checkRole` in `onlyProtocolAdmin`, or use `onlyRole(Roles.protocolAdmin())` instead of this modifier - **Resolved**
11. Instead of creating public getters for private state vars, make them public - **Resolved**
12. Change all calls of `this.function()` to `function` and change function to public, or create an internal function called by the external function. One line functions (like `hasUnderwriter`) can remain external and just use the line itself instead of calling the function within the contract (e.g use `creditLineSettings.borrower` instead of `this.borrower()`) - **Resolved**
13. `repay` could return actual amount repaid for better UX - **Resolved**
14. `interestOwed` in `repay` is unnecessary, can use `_accumulatedInterest`
15. Remove `BORROWER_ROLE` that's never used - **Resolved**
16. Remove `BASIS_POINTS` that's never used in ValidationLogic
17. Use `onlyRole(Roles.protocolAdmin())` modifier in `setEmergencyShutdown` instead of an if-revert
18. `totalDeployed` and `_outstandingDebt` keep track of the same value, should be merged to one var - **Resolved**
19. Update `_lastInterestAccrualTime` to `block.timestamp` in end of `_accrueInterest` (updating from `state.lastAccrualTime` does the same) - **Resolved**
20. no need to check `_underwriter != address(0)` in `_payFeesOnInterestRepayment` since it's already checked in `calculateInterestFees` - **Resolved**
21. Create a new function `_selectRepaymentTranche()` for the shared code between `_executeRepayment` and `_selectDeploymentTranche(amount)` - **Resolved**
22. Remove `tranche.getVirtualBalances()` and instead use `tranche.virtualDeployed` since it's public (all 3 balances are public and can be read directly)
23. separate `_protocolFees` to 3 public vars (or keep and have 3 public getters)
24. in `getAvailableLiquidity()` call `getAvailableForBorrowing` from Tranche (if the tranche is active) and not `getPoolMetrics`
25. The check for `isRegisteredUnderwriter` in `acceptUnderwriter` is redundant since `getUnderwriterFees` already checks this - **Resolved**
26. The check for `_pendingUnderwriter == address(0)` in `acceptUnderwriter` is redundant since the previous check would fail. And same for check for `_underwriter != address(0)` in `rejectTrancheSettingsChange` and `rejectCreditLineSettingsChange` - **Resolved**
27. Create an internal function for repayment used by `repay`, `repayInterest` and `repayAll` to avoid code duplication - **Resolved**

28. Remove `compoundPerSecondRay` that's never used - **Resolved**
29. add `if (totalOwed == 0) revert Errors.AmountMustBeGreaterThanZero();` to `repay`, as in `repayInterest` and `repayAll` (otherwise call succeeds and an event is emitted with 0 amounts) - **Resolved**
30. `x2 * dtSeconds * (dtSeconds - 1))` can be saved in a var and reused in 3rd order calculation in `compoundPerSecondRayBinomial` for efficiency - **Resolved**
31. `validateTrancheSettings` on tranche's `updateSettings` is redundant, same for `validateCreditLineSettings` in `_updateCreditLineSettings` - **Resolved**
32. Remove `interestPaymentInterval` that's not used anywhere
33. `_accrueInterest` in `_updateCreditLineSettings` should only be done in case the interest changes, and not always. - **Resolved**
34. The address args passed to `reject` and `approve` in `SettingsProposal` can be removed (anyway only used with msg.sender) - **Resolved**
35. Reserve's `deployTo` should use `_transfer` (with emitting the same event as withdrawals or taking the event out of the internal function) - **Resolved**
36. In `ProtocolConfiguration`, use openzeppelin's `onlyRole` modifier instead of if-reverts. - **Resolved**
37. `maximumDeposit` in tranche's settings should be named `maximumTVL` to avoid confusion (and also `MAX_DEPOSIT_AMOUNT`,`Errors.ExceedsMaximumDeposit/InvalidMaximumDeposit`) - **Resolved**
38. The `override` keyword is not necessary when inheriting from interface, like in `Reserve` (starting solidity 0.8.8). - **Resolved**
39. `deposit` and `mint` in Tranche should call a `_deposit` function that has the shared code, same for `withdraw` and `redeem` - **Resolved**
40. In `requestWithdraw`, use `maxWithdraw(msg.sender)` instead of `maxUserAssets`
41. `proposeTrancheSettingsChange` can return a boolean signaling whether the settings have been set or they require underwriter approval, for better UX.
42. The check for `request.shares == 0` in `getApprovedShares` is redundant - **Resolved**
43. In `_update`, `recipientSharesBeforeTransfer` and `recipientCheckpoint` are only relevant in case of sending to non-compounding LPs, so they could be created only under `if (toNonCompounding)` and not be conditionally computed. - **Resolved**
44. `getTotalPendingInterest` and `getCurrentInterestDebt` are the same - **Resolved**
45. In `_validateWithdrawalConstraints`, the check of available cash should be done outside of the if statement to avoid code duplication.
46. Instead of `_validateWithdrawalConstraints`, use `maxWithdraw` or `maxRedeem`

47. Some functions are missing event emitting (e.g. `consumeApproval`), make sure all functions that should be tracked are emitting events. -**Example given resolved**

Notes

Some points to consider:

1. Capping the `timeDelta` to 10 years in interest calculation causes interest on debt that hasn't been accrued in over 10 years to stay constant, this requires protocol admin to call `accrueInterest` manually before reaching 10 years.
2. Note that `getProposalInfo` (and so `getPendingTrancheSettingsProposal` and `getPendingCreditLineSettingsProposal`) will return false for `expired` for a proposal that is expired but is not pending. - **Resolved**
3. Note that tranches that are not created by the pool factory can only have withdrawal request functionality if they are registered later in the withdrawalRegistry by a registrar.
4. In `approveRedeemRequest`, the amount of shares approved can be more than the amount requested.
5. Consider whether it is really necessary for the borrowers to be able to call `accrueInterest,` since all the actions they are able to perform (`drawdown, repay/All/Interest)` already accrue the interest. Borrowers can also call `getPendingInterest, getTotalDebt. getCurrentInterestDebt/getTotalPendingInterest a`nd `getDebtDetails i`f they want to know the updated status of their debt.
6. The project is using floating versions for openZeppelin, it's recommended to use fixed versions since newer versions could expose the project to new vulnerabilities or introduce breaking changes.(for example, `ReentrancyGuardUpgradeable` is not in v0.5.5 anymore). Different versions also can result in different bytecode for different people running the code. - **Resolved**
7. Also floating solidity versions are used, consider setting fixed version since it's good practice to prevent that undiscovered vulnerabilities in newer compilers are not added to the project at the time of compiling the source code. - **Resolved**
8. When changing the settings of a tranche to not require withdrawal approvals, there's no resetting of total reserved shares, so the reserved amount will keep being deducted in getAvailableForBorrowing, making it stuck in the reserve and not able to be borrowed, or withdrawn (when withdraw/redeem are fixed to use `getAvailableForBorrowing`). The total reserved for withdrawals will not decrease even if the users that have previously reserved for withdrawals will withdraw the amounts they reserved. To fix: add a function that clears totalReservedShares[tranche] and call it from tranche's updateSettings. Note that users` requests will still be registered in the withdrawal registry and will be ignored when approvals are off, but will be relevant again if approvals are set back on again.

9. Note that `requestWithdraw` is called with amount of assets, but the request is created using the corresponding amount of shares. Later when the user wants to withdraw the same amount of assets, the corresponding amount of shares is calculated again, and if it's bigger than the amount initially requested, the withdrawal will revert. This can happen if interest fees are increased between withdrawal request and withdrawal, since that would increase `netPendingInterest` and thus `TotalAssets` will be smaller. Note that you should display to the users the current amount that they can withdraw.
10. Similarly, If there's been an increase in net pending interest between `requestRedeem` and `redeem`, the amount of assets corresponding to the shares will increase, and the redeem will fail. This is likely to happen since every action on the pool accrues interest, and accrual can also be triggered by borrower or manager. For example: 2 users deposit 500 assets and get 5B shares each and the borrower performs a drawdown of 500 assets, then one user requests to redeem his 5B shares and the request gets approved. At this point the redeem could be performed since the share price hasn't changed. Then an interest of 100 assets is accrued. The totalAssets is now 1100 and the 5B shares are worth 549.95 assets, resulting in the redeeming of those shares to fail, since there's only 500 assets in the tranche.
11. Note that a deactivated tranche remains registered in the withdrawal registry until a registrar calls `revokeTranche` on the withdrawal registry in a separate call. Consider whether you want to keep the withdrawal request mechanism on a deactivated tranche, or automatically cancel it and allow direct withdrawals by changing the `requireApprovalForWithdrawals` setting and revoking the tranche from the withdrawal registry.
12. If funds are sent directly to the reserve, it does not update the virtual balance and so the share price stays the same, and these extra funds can only be withdrawn by an emergency withdrawal. Consider adding a `donate` function to handle donations and record them in the tranche. Also consider adding an admin function to sweep only extra funds over the virtual balance. - **Resolved (with note)**

7. Final Recommendations

Summary

The overall functionality and reliability of the Textile lending protocol have been notably improved with the recent version. v2.1 introduced ERC4626-compliant vault tranches with immutable custody separation (Reserve), factory-based deployment, optional withdrawal approvals, per-second RAY-precision interest compounding, and overall a much more robust protocol.
However, many vulnerabilities identified would not have existed if the codebase had maintained a more straightforward architecture with fewer moving parts and less functionality beyond the protocol's core goals. Overly complex mechanisms not only introduced bugs and inconsistencies but also made both implementation and review much harder.
Careful planning should precede any future development work - minimizing non-essential features and favoring simple structures directly supports better maintainability and auditability.

Remediation & Testing

Due to the large number of issues addressed in this report, it is essential for the review process that each change is done in a different commit, and the commit title clearly directs to the issue it addresses.
Post-remediation testing must confirm recent changes perform as intended, without regressions or new risks added during refactoring. All critical logic paths, especially those newly simplified, should be subject to scenario-based and adversarial test cases to ensure robust, resilient behavior under edge conditions and high-usage scenarios.

8. Conclusion

The audit of Textile v2.1 has revealed several critical and high-severity vulnerabilities that must be addressed before deployment. The most significant issues involve improper checkpoint updates for non-compounding LPs, incorrect interest calculations, and weaknesses in governance and rounding issues. These findings expose risks to fund safety, protocol integrity, and user trust. While the core contracts demonstrate strong adherence to access control and upgrade safety patterns, the identified flaws require urgent remediation. Once these issues are resolved and thoroughly tested, the Textile lending infrastructure will be well-positioned for production use.

9. Summary - Post Remediation

Node.Security performed a security assessment of Textile Protocol v2.1, focusing on the core lending and ERC4626 tranche contracts responsible for deposits, withdrawals, loan issuance, interest accrual, and reserve custody separation. The review combined manual analysis, automated tooling, threat modeling, and scenario-based testing.

Overall, the system demonstrates strong security fundamentals, including role-based access control, upgrade safeguards, and generally careful fund-flow handling. The audit initially identified several issues that could have led to direct loss or misallocation of funds, primarily around non-compounding LP checkpoint handling and interest/fee rounding edge cases, alongside governance and operational risks.

A total of 28 findings were reported: 3 Critical, 4 High, 11 Medium, 10 Low, and 3 Informational. Textile remediated all Critical and High severity issues, and we verified the fixes as reflected in the final statuses in this report. One Low severity issue remains Partially resolved, and one Informational issue remains Unresolved. We recommend continuing maintaining strong regression coverage, with emphasis on adversarial tests around checkpoint updates, interest accrual timing, and rounding behavior to ensure long-term robustness, and adhering to the recommendations given in this report in future updates.

Given the protocol's financial complexity and the sensitivity of its interest and accounting logic, we strongly recommend additional independent audits prior to mainnet deployment, as well as ongoing reviews for future releases. We also recommend running public or private security competitions and bug bounty programs to further stress-test the system under adversarial conditions and uncover edge cases that may not surface during a single audit. These measures will significantly strengthen the protocol's long-term security posture and resilience.