Bulla Factoring Audit Report

Prepared by LPHAC

Version 1.0

Lead Auditors

0xleastwood

0xkamensec

October 3, 2024

Contents

1 About LPHAC

LPHAC is a Web3 security organization aiming to protect projects and their partners against malicious actors. We aim to provide holistic improvements to a project's security stack in a safe and realiable way.

2 Disclaimer

The LPHAC team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Bulla Network enables the on-chain creation of credit pools for invoice factoring. By adhering to the ERC4626 specification, permissioned depositors can earn interest for facilitating invoice funding. Through these contracts, invoice issuers can factor their receivables, allowing them to receive early payments in exchange for a premium. This integration not only broadens the utility of the Bulla Claim Protocol but also provides a new financial mechanism for liquidity and credit management on-chain.

5 Audit Scope

The audit was started on commit a354fe5 for **4** days and a subsequent fix review was conducted on final commit 0aa634a that went for **1** day.

The following contracts were included as part of the review scope, including any related external dependencies:

```
contracts
 BullaClaimInvoiceProviderAdapter.sol
 BullaFactoring.sol
 BullaFactoringAutomationChecker.sol
 DepositPermissions.sol
 FactoringPermissions.sol
 Permissions.sol
 PermissionsWithAragon.sol
 PermissionsWithSafe.sol
interfaces
 IBullaFactoring.sol
 IInvoiceProviderAdapter.sol
```

6 Executive Summary

Over the course of **5** days, the LPHAC team conducted an audit on the Bulla Factoring smart contracts provided by Bulla Network. In this period, a total of **14** issues were found.

Summary

Project Name	Bulla Factoring
Repository	factoring-contracts
Initial Commit	a354fe5ab78f...
Final Commit	0aa634a84b9a...
Audit Timeline	Sept 16th - Sept 20th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	3
Low Risk	4
Informational	6
Gas Optimizations	0
Total Issues	14

Summary of Findings

[H-1] Vault share manipulation attack by short-circuiting `maxRedeem`	✓ Verified Fix
[M-1] `unfactorInvoice` does not consider partially paid invoice amounts	✓ Verified Fix
[M-2] Fee changes are applied retroactively to active invoices	✓ Verified Fix
[M-3] Interest tax is calculated to be a magnitude larger than the specified `taxBps`	✓ Verified Fix
[L-1] Funded invoices can be re-approved by privileged accounts	✓ Verified Fix
[L-2] Rounding errors in withdrawal allows for asset removal while retaining shares	✓ Verified Fix
[L-3] Only `USDC` is supported as a primary asset for facilitating invoices	✓ Verified Fix
[L-4] Potential mismatch in claim token and `BullaFactoring` asset	✓ Verified Fix
[I-1] Multiple divisions can be simplified into a single operation	✓ Verified Fix
[I-2] `pricePerShare` incorrectly scales shares based on `SCALING_FACTOR`	✓ Verified Fix

[I-3] `convertToShares` incorrectly scales when supply is zero	✓ Verified Fix
[I-4] Code hygiene improvements	✓ Verified Fix
[I-5] Share redemptions are not permissioned	✓ Verified Fix
[I-6] Unnecessary double iteration of pool status arrays	✓ Verified Fix

7 Findings

7.1 High

7.1.1 Vault share manipulation attack by short-circuiting `maxRedeem`

Context: BullaFactoring.sol#L722-L734, BullaFactoring.sol#L699-L709, BullaFactoring.sol#L247-L290

Description: Bulla makes use of implementing the `ERC4626` vault standard to simplify profit sharing across depositors. To protect against typical inflation attacks, internally tracked variables `totalDeposits` and `totalWithdrawals` are stored to help calculate the contract's capital account.

The `_redeem()` function does not fully avoid using manipulatable balances as it utilises `availableAssets()` to determine `maxWithdrawableShares`. In the case where `shares > maxWithdrawableShares`, then all available assets will be sent to the redeemer and `maxWithdrawableShares` shares will be burnt. `maxRedeem()` will return zero when `capitalAccount == 0` which is typically only true if all assets have been used to fund invoices and those same invoices are currently impaired.

This attack can be extended in a much more severe manner when we see that the difference between `totalDeposits` and `totalWithdrawals` grows in the negative direction exactly by total accrued interest amount. Let's say we have `totalWithdrawals = totalDeposits + totalInterestAccrued` when all shares have been redeemed, then when we deposit exactly `totalInterestAccrued` into the vault, `totalDeposits - totalWithdrawals == 0`, however, `calculateRealizedGainLoss()` will track all paid invoices which should also be equal to `totalInterestAccrued`.

However, it does not cover any partially paid invoices, meaning if a partially paid invoice is impaired, anyone can capture this value by short-circuiting the logic in `maxRedeem()` because the entire `approvedInvoices[invoiceId].fundedAmountNet` is marked as a realized loss when it really should be `max(approvedInvoices[invoiceId].fundedAmountNet - invoicesDetails.paidAmount, 0)`. Ultimately, any attempt to withdraw non-zero shares would result in zero shares burnt and `availableAssets()` being sent out to the redeemer.

Proof of Concept:

```
function testWithdrawCausingDOSNoDeposits() public {
    address firstDepositor = makeAddr("firstDepositor");
    address secondDepositor = makeAddr("secondDepositor");

    asset.mint(bob, 1e18);
    vm.startPrank(bob);
    asset.approve(address(bullaFactoring), 1e18);
    asset.transfer(address(bullaFactoring), 1e18);
    vm.stopPrank();

    vm.startPrank(firstDepositor);
    asset.approve(address(bullaFactoring), 1e25);
    bullaFactoring.redeem(1e20, firstDepositor, firstDepositor);
    vm.stopPrank();

    uint256 totalSupply = bullaFactoring.totalSupply();
    assertEq(totalSupply, 0, "Total supply should be 0");

    uint256 totalAssets = bullaFactoring.totalAssets();
    assertEq(totalAssets, 0, "Total assets should be 0");


    uint256 firstDepositAmount = 1e3;

    permitUser(firstDepositor, true, 10*firstDepositAmount);

    vm.startPrank(firstDepositor);
    uint256 firstDepositorShares = bullaFactoring.deposit(firstDepositAmount, firstDepositor);
```

```
    vm.stopPrank();

    vm.startPrank(firstDepositor);
    bullaFactoring.withdraw(1, firstDepositor, firstDepositor);
    vm.stopPrank();
}
```

Recommendation: There are several recommendations for this issue:

1. Avoid adding unnecessarily complexity to the `_redeem()` function. The requirement to cap `maxWithdrawableShares` is unnecessary, users would only be exceeding the required shares if they are burning non-existent shares.
2. Avoid changing how we calculate denominators, either use `totalAssets()`, or use available yield. Try to avoid switching between two different logical methods of profit sharing.
3. If withdrawals should follow the same privileged trusted access control, we should implement the whitelisting functionality. This will reduce likelihood any unprivileged user can exploit potential unforeseen errors.

Bulla: Resolved in PR 81, PR 82, PR 85 and PR 94.

LPHAC: Verified fix. Vault deposit/redeem actions have been heavily refactored. `calculateCapitalAccount()` maintains internal variable tracking under `totalDeposits` and `totalWithdrawals`. Further adherence to OpenZeppelin's `ERC4626` implementation has been made and `totalAssets()` also accounts for deployed capital and reserved target fees.

7.2 Medium

7.2.1 `unfactorInvoice` does not consider partially paid invoice amounts

Context: BullaFactoring.sol#L637-L638

Description: When a factored invoice needs to be undone, the original creditor can refund the contract their owed amount including interest and in return receive back the invoice NFT. The calculation for `totalRefundAmount` does not consider any partially paid amount on the invoice, meaning the creditor may refund additional tokens unnecessarily.

```
function unfactorInvoice(uint256 invoiceId) external {
    ...
    (uint256 trueInterest, uint256 trueProtocolFee, uint256 trueAdminFee) = calculateFees(approval,
↪   daysOfInterestToCharge);
    uint256 totalRefundAmount = fundedAmount + trueInterest + trueProtocolFee + trueAdminFee;

    // Refund the funded amount to the fund from the original creditor
    require(assetAddress.transferFrom(originalCreditor, address(this), totalRefundAmount), "Refund
↪   transfer failed");

    // Transfer the invoice NFT back to the original creditor
    address invoiceContractAddress = invoiceProviderAdapter.getInvoiceContractAddress();
    IERC721(invoiceContractAddress).transferFrom(address(this), originalCreditor, invoiceId);
    ...
}
```

Recommendation: `totalRefundAmount` needs to be adjusted by `invoicesDetails.paidAmount`, making sure that this might exceed all fees because `invoiceDetails.faceValue` includes target fees and an invoice might be unfactored before total interest has been accrued.

Bulla: Resolved in PR 90.

LPHAC: Verified fix. Partial payments are now taken into consideration, including the edge case for where an invoice is only partially funded. In this case, a debtor may have overpaid and need to be refunded some amount.

7.2.2 Fee changes are applied retroactively to active invoices

Context: BullaFactoring.sol#L185-L222

Description: When an invoice is funded, target fees are calculated to determine what amount needs to be kept in the contract to pay for future fees owed. Two fee types can be changed when there is an already active invoice that will cause problems in the contract's accounting, notably `taxBps`, `protocolFeeBps` and `adminFeeBps`.

An increase in these fees will be paid out of the depositor's base capital with no amount being returned because `interestApr` is fixed prior to the invoice being funded. A similar decrease in fees would increase the creditor's kickback amount so this seems to be handled in a manner that does not leave funds unaccounted for.

Recommendation: The solution would be to store `taxBps`, `protocolFeeBps` and `adminFeeBps` within the `InvoiceApproval` struct when the invoice is initially approved. This ensures that the accounting always matches up from when the invoice is funded and until it's end state of impairment or being fully paid.

Bulla: Resolved in PR 89.

LPHAC: `taxBps` is still not being included within the `InvoiceApproval` struct, hence `calculateAccruedProfits()` may show a different value completely if `taxBps` is changed. Allowing for sophisticated depositors to take action before an update to `taxBps` to either extract value or limit their own value dilution.

Bulla: This is not an issue, `taxBps` should not be changed basically ever. The magnitude is also very low. We don't want to optimize for this. And it does not impact factorers. Also for compliance, we would want to apply the desired `taxBps` as soon as it's changed. So we will not implement this.

LPHAC: Verified fix and acknowledged that `taxBps` is rarely changed and will be considered a won't-fix.

7.2.3 Interest tax is calculated to be a magnitude larger than the specified `taxBps`

Context: BullaFactoring.sol#L608-L610

Description: The pool owner takes a portion of all interest accrued by each active invoice and realizes this fully when an invoice has been paid and reconciled. The `calculateTax()` functions takes `taxBps` and converts it to mbps by multiplying it by `1000`. But instead of dividing by `10_000_000`, it returns an amount which is 10x larger than the intended result.

```
function calculateTax(uint256 amount) internal view returns (uint256) {
    return Math.mulDiv(amount, taxBps * 1000, 1_000_000);
}
```

Take 1% tax as an example then `taxBps = 100` and `taxBps * 1000 = 100_000`. Dividing by `1_000_000` gives `0.1 = 10%` which is obviously incorrect.

Recommendation: Update the `calculateTax()` function to the following:

```
function calculateTax(uint256 amount) internal view returns (uint256) {
    return Math.mulDiv(amount, taxBps * 1000, 10_000_000);
}
```

Bulla: Resolved in PR 79.

LPHAC: Verified fix. The suggested update to the `calculateTax()` function was applied.

7.3 Low

7.3.1 Funded invoices can be re-approved by privileged accounts

Context: BullaFactoring.sol#L157-L177

Description: The `approveInvoice()` function does not check for an existing `invoiceId` before overwriting storage. This will overwrite critical accountancy values such as `fundedTimestamp` affecting the calculation of interest and fees.

Proof of Concept: The following test can be added to outline the issue identified:

```
function testApproveFundThenReapproveInvoice() public {
    uint256 initialDeposit = 900;
    vm.startPrank(alice);
    bullaFactoring.deposit(initialDeposit, alice);
    vm.stopPrank();

    vm.startPrank(bob);
    uint invoiceIdAmount = 100;
    uint256 invoiceId = createClaim(bob, alice, invoiceIdAmount, dueBy);

    vm.startPrank(underwriter);
    bullaFactoring.approveInvoice(invoiceId, interestApr, upfrontBps, minDays);
    vm.stopPrank();

    (
        bool _approved,
        IInvoiceProviderAdapter.Invoice memory _invoiceSnapshot,
        uint256 _validUntil,
        uint256 _fundedTimestamp,
        uint16 _interestApr,
        uint16 _upfrontBps,
        uint256 _fundedAmountGross,
        uint256 _fundedAmountNet,
        uint16 _minDaysInterestApplied,
        uint256 _trueFaceValue
    ) = bullaFactoring.approvedInvoices(invoiceId);

    assertEq(_approved, true, "Invoice should be approved");
    assertEq(_invoiceSnapshot.debtor, alice, "Debtor should match");
    assertEq(_invoiceSnapshot.creditor, bob, "Creditor should match");
    assertEq(_interestApr, interestApr, "Interest apr should match");
    assertEq(_fundedTimestamp, 0, "_fundedTimestamp should match");
    assertEq(_upfrontBps, upfrontBps, "upfrontBps should match");
    assertEq(_minDaysInterestApplied, minDays, "_minDaysInterestApplied should match");

    console.log("interestApr", _interestApr);

    vm.startPrank(bob);
    bullaClaimERC721.approve(address(bullaFactoring), invoiceId);
    bullaFactoring.fundInvoice(invoiceId, upfrontBps);
    vm.stopPrank();

    vm.warp(block.timestamp + 5 days);
    uint16 newRate = 1000;
    uint16 newUpfrontBps = upfrontBps;
    uint16 newMinDays = 2*minDays;

    vm.startPrank(underwriter);
    vm.expectRevert(); // Would expect this to revert otherwise fundedTimestamp can change back to zero
    bullaFactoring.approveInvoice(invoiceId, newRate, newUpfrontBps, newMinDays);
    vm.stopPrank();
```

```
    (
            _approved,
        _invoiceSnapshot,
        _validUntil,
        _fundedTimestamp,
        _interestApr,
        _upfrontBps,
        _fundedAmountGross,
        _fundedAmountNet,
        _minDaysInterestApplied,
        _trueFaceValue
    ) = bullaFactoring.approvedInvoices(invoiceId);
    assertEq(_approved, true, "Invoice should be approved");

    assertEq(_approved, true, "Invoice should be approved");
    assertEq(_invoiceSnapshot.debtor, alice, "Debtor should match");
    assertEq(_invoiceSnapshot.creditor, bob, "Creditor should match");
    assertEq(_interestApr, newRate, "Interest apr should match");
    assertEq(_fundedTimestamp, 0, "_fundedTimestamp should match");
    assertEq(_upfrontBps, newUpfrontBps, "upfrontBps should match");
    assertEq(_minDaysInterestApplied, newMinDays, "_minDaysInterestApplied should match");
}
```

Recommendation: Add checks to ensure that `invoiceId` is not approved for existing `approvedInvoices`. This could be done by either `approvedInvoices[invoiceId].approved != true` or checking that the `invoiceSnapshot.creditor != address(bullaFactoring)`.

Bulla: Resolved in PR 83.

LPHAC: Verified fix. `approveInvoice()` now checks if an invoice has been funded by checking if it has already been transferred to the `BullaFactoring` contract.

7.3.2 Rounding errors in withdrawal allows for asset removal while retaining shares

Context: BullaFactoring.sol#L314, Math.sol#L228-L230

Description: When profit is accrued by the vault, `convertToShares()` will decrement shares to asset. The vault may end up in a state where `scaledCapitalAccount > assets * SCALING_FACTOR * supply` which would lead to shares being rounded down to zero. In `withdraw()` we will then withdraw `1 asset`, whilst burning zero shares.

We left this issue as a low severity due to the difficulty to properly amplify or compound the rounding error. Instead, scalping profits appears to be limited to `1 wei` asset amounts (assuming profits accrue slowly and profits from invoice unfactoring and other methods is never exceeding more than 100% of the deposited asset amount).

Recommendation: Preferred approach for vault related contracts is to use the `mulDiv()` function that enables rounding direction.

Bulla: Resolved in PR 82.

LPHAC: Verified fix. The suggested change was implemented, `mulDiv()` now makes use of directional rounding.

7.3.3 Only `USDC` is supported as a primary asset for facilitating invoices

Context: BullaFactoring.sol#L379, BullaFactoring.sol#L583, BullaFactoring.sol#L641, BullaFactoring.sol#L784-L814, BullaFactoring.sol#L898

Description: While Bulla only intends to support `USDC` as the target asset being transferred through their smart contracts, other assets may be utilised in the future and many tokens do not simply return boolean values on successful transfer. `USDT` for example does not return anything and hence all transfers will revert because Bulla checks the return value for a true `success` result.

Recommendation: Make use of OpenZeppelin's `SafeERC20` library to facilitate asset transfers within the `BullaFactoring` and periphery contracts.

Bulla: Resolved in PR 87.

LPHAC: Verified fix. All asset transfers now make use of OpenZeppelin's `SafeERC20` library.

7.3.4 Potential mismatch in claim token and `BullaFactoring` asset

Context: BullaFactoring.sol#L157-L177

Description: The Bulla claim contract manages invoice generation and is generalised when it comes to the asset type used for invoice payments. However, the Bulla factoring contract supports only one asset and therefore if multiple assets are intended to be supported, new instances of the same contract will be deployed.

However, the `invoiceProviderAdapter` is likely the same across all pools and therefore there needs to be some validation of the invoice payment token and the pool's designated asset. Otherwise, it may be possible to approve/fund an invoice in one asset and pay the claim using another asset.

Recommendation: Check that the claim token is the same as the factoring contract's `assetAddress` before approving an invoice.

Bulla: Resolved in PR 84.

LPHAC: Verified fix.

7.4 Informational

7.4.1 Multiple divisions can be simplified into a single operation

Context: BullaFactoring.sol#L233, BullaFactoring.sol#L429

Description: Multiple divisions can be simplified into a single operation, reducing gas usage and increasing readability.

The affected instances occur in the following lines:

1. `(block.timestamp - approval.fundedTimestamp) / 60 / 60 / 24 : 0;`
2. `uint256 daysUntilDue = (invoice.dueDate - block.timestamp) / 60 / 60 / 24;`

Recommendation: Both can be simplified using solidity's `1 days` supported alias.

Bulla: Resolved in PR 88.

LPHAC: Verified fix.

7.4.2 `pricePerShare` incorrectly scales shares based on `SCALING_FACTOR`

Context: BullaFactoring.sol#L301

Description: The function `pricePerShare()` incorrectly scales `capitalAccount`, without following the behaviour elsewhere by dividing by that scaled amount. This means that `pricePerShare > 1e6 * truePricePerShare` where `truePricePerShare` would be the actual real world price of `1 share` in assets.

`1 asset` is always `1 asset`, we only need to scale decimals when comparing different assets that have different decimals, or when using virtual shares which involves scaling up purchased shares by some multiplier (typically `1e3 - 1e5`). In the latter case, it would be wise to scale down `pricePerShare()`, since shares are scaled up by some factor in `deposit()`.

Recommendation: Remove the scaling, or add it to the division operation as well for consistency.

LPHAC: The Bulla team has removed the scaling altogether and hence this issue has been indirectly addresses and therefore can be marked as resolved.

7.4.3 `convertToShares` incorrectly scales when supply is zero

Context: BullaFactoring.sol#L310

Description: Function `convertToShares()` incorrectly scales by `SCALING_FACTOR` only in cases where `supply == 0`. This isn't exploitable since `convertToShares()` is only used in withdrawal, and withdrawing non-zero shares when `supply == 0` is not possible.

Recommendation: Consider removing scaling, or consistently applying it to avoid scenarios where calculations might be unexpectedly scaled.

Bulla: Resolved in PR 78.

LPHAC: Verified fix. `convertToShares()` no longer scales up by `SCALING_FACTOR` when `supply == 0`.

7.4.4 Code hygiene improvements

Context: BullaFactoring.sol#L207, BullaFactoring.sol#L273-L276

Description: Code hygiene improves readability for developers and auditors. The following ares could lead to confusion;

1. Line 207: Magic number `1000_0000` used, this is difficult to read and can cause issues in future development if misread. Recommend using `10_000_000`.
2. Function `calculateRealizedGainLoss()` iterates through the same array twice unnecessarily between lines 258-261 and then 273-276. Recommend simplifying and minimising for loop iteration. Consider further amendments to this function by removing dead invoices out of the array we are iterating through for further efficiency improvements as well as preventing out of gas reverts when block gas limits are close to exhaustion.
3. There are several places (i.e. in the `redeem()` and `withdraw()` functions) where `owner` shadows `StateVar Ownable.owner`. It is advised not to shadow global variables as compiler errors may be introduced and unexpected behaviour may occur.

Recommendation: Consider alterations where appropriate.

Bulla: Resolved in PR 94, PR 95 and PR 96.

LPHAC: Verified fix.

7.4.5 Share redemptions are not permissioned

Context: BullaFactoring.sol#L859-L873

Description: `deposit()` is a permissioned function which limits interaction with the protocol from unapproved users. However, transfers are enabled to any account, thus any unapproved user could be the resultant withdrawer. It is worth noting that this behaviour is known by the team, and whether it is acceptable or not is still being determined.

Proof of Concept: Below is a test outlining the specific behaviour:

```
function testApprovedDepositUnapprovedWithdrawal() public {
    address firstDepositor = makeAddr("firstDepositor");
    address unapprovedWithdrawer = makeAddr("unapprovedWithdrawer");

    uint256 firstDepositAmount = 1e8;

    permitUser(firstDepositor, true, 10*firstDepositAmount);

    vm.startPrank(firstDepositor);
    uint256 firstDepositorShares = bullaFactoring.deposit(firstDepositAmount, firstDepositor);
    bullaFactoring.transfer(unapprovedWithdrawer, firstDepositorShares);
    vm.stopPrank();

    vm.startPrank(unapprovedWithdrawer);
    bullaFactoring.redeem(bullaFactoring.balanceOf(unapprovedWithdrawer), unapprovedWithdrawer,
↪   unapprovedWithdrawer);
    vm.stopPrank();

    uint256 totalSupply = bullaFactoring.totalSupply();
    assertGt(totalSupply, 0, "Total supply should be greater than 0 due to unapproved withdrawals");
}
```

Recommendation: Consider if transferability of depositor shares is applicable relative to your access control policy, if it isn't, consider implementing a withdrawer whitelist.

Bulla: Resolved in PR 85, PR 86 and PR 93.

LPHAC: Verified fixes. Both `_msgSender()` and `_owner` must now have deposit permissions to be able to withdraw/redeem shares.

7.4.6 Unnecessary double iteration of pool status arrays

Context: BullaFactoring.sol#L479-L521

Description: The `viewPoolStatus()` function is used internally when reconciling paid invoices to realize profit and fees, updating the accounting accordingly. By iterating through all active invoices, we do not know ahead of time which invoices are paid or impaired or if previously impaired invoices are now paid. Therefore Bulla is optimistic about the array size and then copies this array to a smaller array when the size is known.

However, this can be done more simply with significant optimisation by overwriting the length parameter of the memory array.

Recommendation: Consider making this change to simplify the `viewPoolStatus()` function significantly.

Bulla: Resolved in PR 88.

LPHAC: Verified fix. The change was made to override the array length parameter using assembly.