Lumin Security Review

Pashov Audit Group

Conducted by: pashov

November 1st, 2023

Contents

1. About pashov

Krum Pashov, or **pashov**, is an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Check his previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Lumin** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Lumin

Lumin is a fixed-rate peer-to-peer lending protocol that will be deployed on multiple EVM-compatible chains, starting with Arbitrum. There are lending, borrowing, repayment and liquidation mechanisms in place.

Any user can deposit funds into the Lumin platform. They can offer loans with them (with a fixed interest rate) or use them as collateral to get a loan themselves as borrowers.

A borrower calls `LoanManager::createLoan`, providing the data for the lender's loan configuration, amount of asset borrowed, the price feed chosen and the collateral assets with their price feeds chosen.

More docs

Observations

The codebase is using upgradeability and pausability which is a centralization attack vector. It is especially dangerous as the protocol handles user allowances which in combination with upgradeability can be used to exploit user allowances.

Collateral assets go through a whitelisting process by the Asset admin through `AssetManager`. Each token integrated should be carefully tested for corner case scenarios - for example Aave tokens are pausable, so can be problematic for liquidations.

The protocol at its current state is incomplete, for example liquidation fees can't be claimed out of the `LoanManager` contract.

The platform allows deposit/collateral usage of ERC20 assets only.

Loans can be liquidated not only when they are "underwater" but also when interest/principal payments are past due date or when the loan term has completed.

Loan and configuration IDs start from 1, not 0 in the platform.

Privileged Roles & Actors

- Lumin admin role - can upgrade, pause and unpause the `AssetManager` and `LoanManagerDelegator` contracts
- Asset Admin role - can manage allowed assets and their price feeds
- Loan Manager role - can do all types of actions with user assets - lock, unlock, lend, seize and more. Should only be given to the `LoanManagerDelegator` contract. Can also mint/burn loan shares
- Any user - can deposit (provide liquidity) and withdraw his non-locked assets from the `AssetManager`
- Lender - creates a configuration to give a loan, then delete it or enable/disable it
- Borrower - creates (takes) a new loan, then repay the interest/principal (anyone can do repays for any borrower)
- Liquidator - liquidates a loan, possibly receiving a reward

5. Risk Classification

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

5.1. Impact

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

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

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

6. Security Assessment Summary

review commit hash **- a4c32f0ad8dd8e424587b1fa10003603e2088a44**

fixes review commit hash **- 3c10ad12db9906bdea2dbf9ebad035f81b190b9e**

Scope

The following smart contracts were in scope of the audit:

- `interfaces/**`
- `access/LuminAccessManager`
- `access/Roles`
- `asset/AssetManager`
- `loan/LoanConfigManager`
- `loan/LoanShares`
- `loan/LoanManager`
- `loan/LoanManagerBase`
- `loan/LoanManagerDelegator`
- `pricefeed/PriceFeedProxyChainlink`

7. Executive Summary

Over the course of the security review, pashov engaged with Lumin to review Lumin. In this period of time a total of **16** issues were uncovered.

Protocol Summary

Protocol Name	Lumin
Date	November 1st, 2023

Findings Count

Severity	**Amount**
Critical	1
High	1
Medium	5
Low	9
Total Findings	**16**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Collateral double-spend post liquidation is possible	Critical	Resolved
[H-01]	Disabled lender's loan configuration can be used by a borrower	High	Resolved
[M-01]	Asset loan config can never be removed from its EnumerableSet	Medium	Resolved
[M-02]	Instant liquidations can happen after unpausing the protocol	Medium	Resolved
[M-03]	Price feed oracle data validation is missing	Medium	Resolved
[M-04]	Lumin admin can exploit user allowances to the protocol	Medium	Resolved
[M-05]	Protocol does not support fee-on-transfer and rebasing tokens	Medium	Resolved
[L-01]	Missing check if priceFeedProxy is set	Low	Resolved
[L-02]	Asset uniqueness should not be based on its symbol	Low	Resolved
[L-03]	Explicitly limit the max number of LoanShares with the same ID	Low	Resolved
[L-04]	A code path is missing asset existence check	Low	Resolved
[L-05]	Protocol can't be deployed with default configuration	Low	Resolved
[L-06]	Using address(0) as a valid value	Low	Resolved
[L-07]	User paying too much interest won't get a refund	Low	Resolved

[L-08]	Code is not following the Checks-Effects-Interactions pattern	Low	Resolved
[L-09]	User can take a loan from himself	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Collateral double-spend post liquidation is possible

Severity

Impact: High, as a borrower can double-spend his collateral

Likelihood: High, as liquidations never subtract the collateral amount

Description

In `AssetManager::assetTransferOnLoanAction` in the `if (action == AssetActionType.Seize)` statement, the transferred amount is only subtracted from `userDepositFrom.lockedAmount`, but it should have also been subtracted from `userDepositFrom.depositAmount` as well. This means that a borrower's collateral balance won't be decreased on liquidation, even though the loan shareholders will receive most of the collateral.

Here is an executable Proof of Concept unit test that demonstrates the vulnerability (you can add this in the end of your `LoanManager.Repay` test file):

```
function test_LiquidateExploit() public {
    // because of the `setUp` method, at this point Bob has taken a loan from
    // Alice through Lumin
    uint256 bobDeposit = wrappedAssetManager.depositOf
      (assetId[1], bob).depositAmount;

    // make loan expire
    vm.warp(block.timestamp + 300 days + 1);
    // random user liquidates Bob's loan, so collateral
    //(asset[1]) should be transferred to the lender
    vm.prank(0x1111111111111111111111111111111111111111);
    wrappedLoanManagerDelegator.liquidate(1);

    // Bob can withdraw his whole deposit even though part of it was used as
    // collateral and was liquidated
    uint256 bobCollateralWalletBalance = mockERC20Token[1].balanceOf(bob);
    vm.prank(bob);
    wrappedAssetManager.assetDepositWithdraw
      (assetId[1], IAssetManager.AssetActionType.Withdraw, bobDeposit);

    assertEq(mockERC20Token[1].balanceOf
      (bob) - bobCollateralWalletBalance, bobDeposit);
}
```

Recommendations

Change the code in `assetTransferOnLoanAction` in the following way:

```
else if (action == AssetActionType.Seize) {
        userDepositFrom.lockedAmount -= amount;
+       userDepositFrom.depositAmount -= amount;
        userDepositTo.depositAmount += amount;
    }
```

8.2. High Findings

[H-01] Disabled lender's loan configuration can be used by a borrower

Severity

Impact: Medium, as a borrower can use a disabled loan configuration but it will still work as a normal loan, so no lender value loss

Likelihood: High, as the disabling functionality can never work with the current code

Description

In the `LoanConfig` struct we have the `enabled` field, which is set to `true` for newly created loan config and can be set to `true/false` in `LoanConfigManager::updateLoanConfigEnabledStatus`. The problem is that in `LoanManager::createLoan`, when a borrower takes in a loan from a lender, a `configId` is given as an argument and then the corresponding `LoanConfig` struct object is used in the method, but the `enabled` property is never checked. This means that even when a lender has called `LoanConfigManager::updateLoanConfigEnabledStatus` with `enabled == false` for a loan configuration they created, the configuration can still be used by borrowers.

Here is an executable Proof of Concept unit test that demonstrates the vulnerability (you can add this in the end of your `LoanManager` test file):

```
function test_DisabledLoanConfigCanStillBeUsed() public {
    // Alice disables her loan config
    vm.prank(alice);
    wrappedLoanManagerDelegator.updateLoanConfigEnabledStatus
      (loan.configId, false);
    vm.startPrank(bob);

    // there are no current loans
    assertEq(0, wrappedLoanManagerDelegator.getLoanCounter());

    vm.expectEmit();
    emit LoanCreated(1, 1);
    wrappedLoanManagerDelegator.createLoan(loan, collateralAssets);

    // a loan was created
    assertEq(1, wrappedLoanManagerDelegator.getLoanCounter());

    // Alice's free deposit was lowered

    assertEq(
      aliceDepositAfterLending.depositAmount,
      userDepositAlice[0]-loan.principalAmount
    );
    assertEq(aliceDepositAfterLending.lockedAmount, 0);
}
```

Recommendations

Check if a loan config is enabled in `LoanManager::createLoan` and revert the transaction if it is not.

8.3. Medium Findings

[M-01] Asset loan config can never be removed from its EnumerableSet

Severity

Impact: Medium, as a user will be limited in his platform usage but won't lose value

Likelihood: Medium, as it happens when a user has created the max total loan configs for an asset

Description

The `LoanConfigManager` contract uses the `userConfigIds` mapping to keep track of the count of loan configurations a user created for an asset and limit them up until `maxLoanConfigsPerAsset` (which is set to 10 in the `LuminDeploy` deployment script). The problem is with the `deleteLoanConfig` method, which looks like this:

```
function deleteLoanConfig(uint256 configId) external {
    LoanConfig storage config = loanConfigs[configId];
    if (config.lender != msg.sender) {
        revert NotAuthorized();
    }

    if (config.totalPendingPrincipalAmount > 0) {
        revert LoanConfigInUse();
    }

    delete (loanConfigs[configId]);
    userConfigIds[msg.sender][config.config.principalAssetId].remove(configId);

    emit LoanConfigDeleted(configId);
}
```

The issue here is quite subtle - on the line using `userConfigIds`, where the `config` storage pointer is used, the value in the storage cells it points to is already zeroed-out because of the previous `delete` command, which deletes the storage slots to which the `config` pointer points to. This means that no actual removal will be executed, since `config.config.principalAssetId` will

be 0 and the `EnumerableSet::remove` method does not revert when item to remove is not in the set. This means that the set in `userConfigIds` can only grow, but since it is limited to 10 items at a certain point the user will hit this check in `createLoanConfig`:

```
if (userConfigIds[msg.sender][config.principalAssetId].length
  () >= limits.maxLoanConfigsPerAsset) {
    revert MaxConfigsReached();
}
```

This means that the user won't be able to add more loan configurations for a given asset any more, no matter what he does.

Here is an executable Proof of Concept unit test that demonstrates the vulnerability (you can add this in the end of your `LoanConfigManager` test file):

```
function test_AssetLoanConfigIsNotRemovedFromEnumerableSet() public {
    uint256 amount = 100_000;

    vm.startPrank(alice);
    mockERC20Token[0].approve(address(assetManagerProxy), amount);
    wrappedAssetManager.assetDepositWithdraw(
      loanConfigUser.principalAssetId,
      IAssetManager.AssetActionType.Deposit,
      amount
    );

    // Create 10
    //(the maximum total allowed for a single asset) loan configs for the same asset
    for (uint256 i = 0; i < 10; i++) {
        wrappedLoanManagerDelegator.createLoanConfig
          (loanConfigUser, loanConfigAssetUsage);
    }

    // Remove one of the loan configs
    wrappedLoanManagerDelegator.deleteLoanConfig(1);

    // Try to create another loan config for the same asset, but
    // it reverts since the count
    //(length of the EnumerableSet) wasn't decremented
    vm.expectRevert(abi.encodeWithSelector
      (ILoanConfigManager.MaxConfigsReached.selector));
    wrappedLoanManagerDelegator.createLoanConfig
      (loanConfigUser, loanConfigAssetUsage);
}
```

Recommendations

Change the code in the following way:

```
function deleteLoanConfig(uint256 configId) external {
    LoanConfig storage config = loanConfigs[configId];
    if (config.lender != msg.sender) {
        revert NotAuthorized();
    }

    if (config.totalPendingPrincipalAmount > 0) {
        revert LoanConfigInUse();
    }

-    delete (loanConfigs[configId]);
-    userConfigIds[msg.sender][config.config.principalAssetId].remove(configId);
+    userConfigIds[msg.sender][config.config.principalAssetId].remove(configId);
+    delete (loanConfigs[configId]);

    emit LoanConfigDeleted(configId);
}
```

And also, as a best practice, always check the `remove` method's return value and revert if equals `false`.

[M-02] Instant liquidations can happen after unpausing the protocol

Severity

Impact: High, as users can be instantly liquidated and lose value

Likelihood: Low, as it requires a long pause from the protocol admin

Description

The `LoanManagerDelegator` contract through which all protocol interactions happen is pausable by the Lumin admin. In the case that the protocol is paused for a long time, borrowers' collateral assets can fall in price and their loans might become liquidateable without a way for them to repay them or to add collateral, or even their loan term can pass. This means when the protocol is unpaused the loan can get instantly liquidated resulting in a loss for the borrower.

Recommendations

Add a post-unpause grace period for liquidations to give time for users to repay their loans or add collateral.

[M-03] Price feed oracle data validation is missing

Severity

Impact: High, as it can possibly result in unfair liquidations

Likelihood: Low, as it only happens in specific rare conditions

Description

There are multiple problems when validating the price feed data in `PriceFeedProxyChainlink`:

- the code doesn't revert when `intPrice == 0`
- price staleness is not checked
- the `GRACE_PERIOD` after a sequencer was down is not waited

Using an incorrect price can be detrimental for the protocol as it can lead to unfair loans/liquidations.

Recommendations

For the sequencer `GRACE_PERIOD` make sure to follow the Chainlink docs. Also iplement the following change:

```
- if (intPrice < 0) {
+ if (intPrice <= 0) {
    revert ImplausiblePrice();
}
```

Finally, check the timestamp value of the `latestRoundData` call and make sure it hasn't been longer than the heartbeat interval for the price feed (plus 10-30 minutes buffer period).

Discussion

pashov: Partially fixed, excluding the price staleness check.

[M-04] Lumin admin can exploit user allowances to the protocol

Severity

Impact: High, as users can get their allowance stolen

Likelihood: Low, as it requires a malicious or compromised owner

Description

The `AssetManager` contract is used by users to deposit assets into the platform by giving allowance to the contract to execute an `ERC20::transferFrom` call. The problem is that the contract is upgradeable, meaning the Lumin admin can back-run a user approval to the `AssetManager` with an upgrade that adds functionality to execute a `transferFrom` from the user to his address through the contract.

Recommendations

Put the Lumin Admin role holder address to be behind a Timelock contract so that users can react to admin actions.

[M-05] Protocol does not support fee-on-transfer and rebasing tokens

Severity

Impact: High, as this can leave tokens stuck in the protocol

Likelihood: Low, as a small portion of the commonly used tokens have such mechanisms

Description

The `_depositWithdraw` method in `AssetManager` has the following implementation:

```
function _depositWithdraw
  (address assetAddress, bool deposit, address sender, uint256 amount) private {
    if (deposit) {
        IERC20(assetAddress).safeTransferFrom(sender, address(this), amount);
    } else {
        IERC20(assetAddress).safeTransfer(sender, amount);
    }
}
```

Also before/after it is called we have code like this:

```
assetDeposit.depositAmount -= amount;
userDeposit.depositAmount -= amount;
```

and this

```
assetDeposit.depositAmount += amount;
userDeposit.depositAmount += amount;
```

This code does not account for tokens that have a fee-on-transfer or a rebasing (token balance going up/down without transfers) mechanisms. By caching (or removing) the `amount` given to the `transfer` or `transferFrom` methods of the ERC20 token, this implies that this will be the actual received/sent out amount by the protocol and that it will be static, but that is not guaranteed to be the case. If fee-on-transfer tokens are used, on deposit action the actual received amount will be less, so withdrawing the same balance won't be possible. For rebasing tokens it is also possible that the contract's balance decreases over time, which will lead to the same problem as with the fee-on-transfer tokens, and if the balance increases then the reward will be stuck in the `AssetManager` contract.

Recommendations

You can either explicitly document that you do not support tokens with a fee-on-transfer or rebasing mechanism or you can do the following:

For fee-on-transfer tokens, check the balance before and after the transfer and validate it is the same as the `amount` argument provided. For rebasing tokens, when they go down in value, you should have a method to update the cached reserves accordingly, based on the balance held. This is a complex solution. For rebasing tokens, when they go up in value, you should add a method to actually transfer the excess tokens out of the protocol (possibly directly to users).

8.4. Low Findings

[L-01] Missing check if `priceFeedProxy` is set

In `AssetManager::updatePriceFeedProxyEnableStatus` you should check if `priceFeedProxies[index] == address(0)` and revert the execution if that's the case, since that would mean setting the enable status of a not yet set proxy.

[L-02] Asset uniqueness should not be based on its symbol

In `AssetManager::addAsset` the data to calculate the an asset's ID is currently the `symbol` argument value (plus the hardcoded 0 value for `collectionId`). This is suboptimal and can possibly limit your application, as when generating ID by hashing some data, you'd want that data to be unique. Symbols (NFT symbols, ERC20 token symbols) are not guaranteed to be unique and two different tokens/collections can have the same symbol. The correct way to generate the ID from unique data is to use the `assetAddress` argument value (in combination with `collectionID` for ERC721/ERC1155 collections) since addresses are unique and can't be the same for different tokens/collections.

[L-03] Explicitly limit the max number of `LoanShares` with the same ID

The `LoanShares` contract is an implementation of ERC1155 meaning there can be multiple items with the same ID. Currently, the `mint` method always mints `NUM_SHARES_PER_LOAN` (hardcoded at 10) for a given ID, but it doesn't restrict multiple mints for the same ID. Since the `burn` and `sharesOf` methods of `LoanShares` call `nftOwners[id].values()`, which iterates over all items for a given ID, you should explicitly limit the max number of items to be minted with the same ID to 10, so that you do not get to a DoS state because of array iteration in the `EnumberableSet::values` call.

[L-04] A code path is missing asset existence check

The `assetTransferOnLoanAction` method in `AssetManager` goes through different code paths based on the `action` argument it received. If `action == AssetActionType.Lend` then it calls `_assetAction`, which has the `assetExists` modifier. The problem is that in all other cases, for example when `action == AssetActionType.Lend` asset existence is not checked. Make sure to add asset existence check for all code paths in `assetTransferOnLoanAction`.

[L-05] Protocol can't be deployed with default configuration

The contracts in the protocol are using compiler version 0.8.20 and the docs specifically say that the protocol will be deployed on Arbitrum. While Arbitrum is EVM-compatible, the PUSH0 opcode is still not supported on the network, and the 0.8.20 compiler version produces bytecode that contains the PUSH0 opcode. This will result in a failure when trying to deploy the contracts to Arbitrum (or other such L2 networks). Make sure to use Solidity compiler version 0.8.19.

[L-06] Using `address(0)` as a valid value

In `AssetManager::_assetAction` we can see the following line of code:

```
DepositData storage assetDeposit = deposits[assetId][address(0)];
```

This is used to track the total deposits by any user for an asset (also special handling is implemented in `AssetManager::balancesOf`). The problem is that `address(0)` is used as a valid value here, but `address(0)` is also the default value for an `address` typed variable or storage slot. This is error-prone and can lead to bugs where the default value of the `deposits` mapping is used and it is non-empty, especially given that the `assetTransferOnLoanAction` has no zero-address checks for the `userAddressFrom` or `userAddressTo` argument values. As a best practice, restrain from using `address(0)` as a valid value by possibly using a separate mapping.

[L-07] User paying too much interest won't get a refund

The `pay` method of `LoanManager` explicitly says in its NatSpec docs:

> /// @dev Does not refund when too much interest is paid.

This means that when a borrower repays more interest than what he owes, the excess value will be given to the lender. While this is explicitly mentioned as part of the design, this can still lead to a value loss for the borrower (if he input the wrong value for the `interestAssetAmount` argument in `pay`) which shouldn't be possible. I suggest holding this value in a mapping which is claimable by the borrower.

[L-08] Code is not following the Checks-Effects-Interactions pattern

The `createLoan` method in `LoanManager` mints ERC1155 tokens, but they do an unsafe call (the `onERC1155Received` call) to the token receiver. This call can reenter the `LoanManager` contract while it is in pending state change (the `assetTransferOnLoanAction` calls haven't been executed at this point). Make sure to move the `mint` call to be the last one in the `createLoan` method to follow the CEI pattern.

[L-09] User can take a loan from himself

The `LoanManager` contract allows a user to take a loan from himself. This has no functional value for the protocol but widens the attack surface for a potential vulnerability. Add a check to revert when the same address that created a loan config is using it to take out a loan from the system.