RENASCENCE

Arcadia Finance Audit Report

Version 2.0

Audited by:

HollaDieWaldfee

alexxander

bytes032

January 12, 2024

Contents

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

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

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality
- Low - Funds are **not** at risk

1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

2 Executive Summary

2.1 About Arcadia Finance

Arcadia is a non-custodial platform for asset management focused on unlocking the full potential of DeFi.

The platform gives users a single on-chain account to access margin, compose a portfolio of multiple yield sources, and improve capital efficiency across underlying DeFi protocols.

Arcadia V2 introduces several new features:

- An enhanced UI/UX for a user-friendly experience
- Portfolio Strategies allowing users to combine different yield sources
- Optimistic Actions for streamlined asset management
- A Yield Optimizer for constructing and rebalancing portfolios
- A Multicall Calldata Generator for executing complex transactions in a single step.

These improvements aim to simplify on-chain strategies, making them accessible to a broader user base.

2.2 Overview

Project	Arcadia Finance
Repository	lending-v2, accounts-v2
Commit	a1886e2d1bf3..., 05cea753c358...
Mitigation	d095799af095..., afe3debe14a3...
Date	18 December 2023 - 8 January 2024

2.3 Issues Found

Severity	**Count**
High Risk	6
Medium Risk	5
Low Risk	9
Informational	3
Total Issues	**23**

3 Findings Summary

ID	Description	Status
H-1	The asset valuation for derived assets is broken	Resolved
H-2	Reentrancy in Account initialization can be used to drain creditors' funds	Resolved
H-3	AccountV1.setApprovedCreditor() doesn't trigger a transfer delay and can be used to steal funds through AccountV1.flashActionByCreditor()	Resolved
H-4	Reentrancy during liquidation corrupts the LendingPool accounting completely	Resolved
H-5	Liquidation of unhealthy accounts can be prevented at low cost	Resolved
H-6	Misjudgment regarding reentrancy risks associated with ERC777 tokens can lead to direct theft of funds	Resolved
M-1	Under certain conditions, direct theft of funds is possible during liquidation	Resolved
M-2	Zero account value renders liquidations impossible	Resolved
M-3	UniswapV3AssetModule exposure can be manipulated by withdrawing zero amounts	Resolved
M-4	Calling `batchProcessDeposit()` before `batchProcessWithdrawal()` allows bypassing exposure limits	Resolved
M-5	UniswapV3AssetModule calculates underlying asset amounts irrespective of amount parameter	Resolved
L-1	AccountV1.setAssetManager() should be callable by everyone	Resolved
L-2	LendingPool.liquidityOf() should round down instead of up	Resolved
L-3	settleAuction() doesn't account for fixedLiquidationCost	Resolved
L-4	Tranche interestRate can be used for inflation attack	Resolved
L-5	UniswapV3AssetModule.isAllowed() should check if liquidity is greater than zero	Resolved
L-6	Accounts cannot be liquidated if no Tranches are set	Resolved
L-7	Incorrect interest rate while auctions are ongoing	Acknowledged
L-8	Liquidations of dust debt positions are unprofitable to initiate which can lead to DoS attempts	Resolved
L-9	Asset shares in auctions should be rounded up to the bidder's disadvantage	Resolved
I-1	Withdraw event is emitted with wrong parameter	Resolved
I-2	Remove unused events	Resolved

ID	Description	Status
I-3	Unsafe casts of totalRealisedLiquidity	Resolved

4 Findings

4.1 High Risk

[H-1] The asset valuation for derived assets is broken

Context: Registry.sol, AccountV1.sol

Impact: Derived assets are incorrectly valued which among other things can lead to premature liquidations.

Description: An account's numeraire is the unit in which all assets and liabilities of the Account are measured. The numeraire can be a derived asset and doesn't have to be a simple ERC20.

The numeraire plays a significant part in calculating the liquidation value.

There are two possible ways the liquidation value is calculated, starting from:

- `Registry.getLiquidationValue()`
- `AccountV1.startLiquidation()`

In the Registry's case, the downstream call to get the numeraire value is:

```
## Registry.sol

    function getLiquidationValue(
        address numeraire,
        address creditor,
        address[] calldata assetAddresses,
        uint256[] calldata assetIds,
        uint256[] calldata assetAmounts
    ) external view returns (uint256 liquidationValue) {
        AssetValueAndRiskFactors[] memory valuesAndRiskFactors =
            getValuesInUsd(creditor, assetAddresses, assetIds, assetAmounts);

        // Calculate the "liquidationValue" in USD with 18 decimals precision.
        liquidationValue = valuesAndRiskFactors._calculateLiquidationValue();

        // Convert the USD-value to the value in Numeraire if the Numeraire is
        different from USD (0-address).
        if (numeraire != address(0)) {
>           liquidationValue = _convertValueInUsdToValueInNumeraire(numeraire,
liquidationValue);
        }
    }
```

```
## Registry.sol

    function _convertValueInUsdToValueInNumeraire(address numeraire, uint256
    valueInUsd)
        internal
        view
        returns (uint256 valueInNumeraire)
    {
        // We use the USD price per 10^18 tokens instead of the price per token to
        guarantee sufficient precision.
>        (uint256 rateNumeraireToUsd,,) =
            IAssetModule(assetToAssetModule[numeraire]).getValue(address(0),
            numeraire, 0, 1e18);

        // "valueInUsd" is the USD-value of the assets with 18 decimals precision.
        // "rateNumeraireToUsd" is the USD-value of 10 ** 18 tokens of numeraire with
        18 decimals precision.
        // To get the value of the asset denominated in the numeraire, we have to
        multiply USD-value of "valueInUsd" with 10**18
        // and divide by "rateNumeraireToUsd".
        valueInNumeraire = valueInUsd.mulDivDown(1e18, rateNumeraireToUsd);
    }
```

On the other hand, in the Account's case it is:

```
## AccountV1.sol

    function startLiquidation(address initiator)
        external
        onlyLiquidator
        nonReentrant
        updateActionTimestamp
        returns (
            address[] memory assetAddresses,
            uint256[] memory assetIds,
            uint256[] memory assetAmounts,
            address creditor_,
            uint256 openPosition,
            AssetValueAndRiskFactors[] memory assetAndRiskValues
        )
    {
        inAuction = true;
        creditor_ = creditor;

        (assetAddresses, assetIds, as.setAmounts) = generateAssetData();
        assetAndRiskValues =
>            IRegistry(registry).getValuesInNumeraire(numeraire, creditor_,
assetAddresses, assetIds, assetAmounts);
```

Here, it can be observed that the `creditor` is passed as the first argument, whereas in the Registry's case, it was `address(0)`.

```
## Registry.sol

    function getValuesInNumeraire(
        address numeraire,
        address creditor,
        address[] calldata assetAddresses,
        uint256[] calldata assetIds,
        uint256[] calldata assetAmounts
    ) external view returns (AssetValueAndRiskFactors[] memory valuesAndRiskFactors) {
        valuesAndRiskFactors = getValuesInUsd(creditor, assetAddresses, assetIds,
        assetAmounts);

        // Convert the USD-values to values in numeraire if the numeraire is different
        from USD (0-address).
        if (numeraire != address(0)) {
            // We use the USD price per 10^18 tokens instead of the price per token to
            guarantee sufficient precision.
>            (uint256 rateNumeraireToUsd,,) =
>                IAssetModule(assetToAssetModule[numeraire]).getValue(creditor,
numeraire, 0, 1e18);

            uint256 length = assetAddresses.length;
            for (uint256 i; i < length; ++i) {
                // "valuesAndRiskFactors.assetValue" is the USD-value of the asset
                with 18 decimals precision.
                // "rateNumeraireToUsd" is the USD-value of 10 ** 18 tokens of
                numeraire with 18 decimals precision.
                // To get the asset value denominated in the numeraire, we have to
                multiply USD-value of "assetValue" with 10**18
                // and divide by "rateNumeraireToUsd".
                valuesAndRiskFactors[i].assetValue =
                    valuesAndRiskFactors[i].assetValue.mulDivDown(1e18,
                    rateNumeraireToUsd);
            }
        }
    }
```

This doesn't make a difference when `numeraire` is an asset from `StandardERC20AssetModule`. But it does make a difference for derived assets.

Downstream in the pricing of the derived numeraire we might completely set its value to zero because it doesn't meet `minUSDValue_`.

```
## Registry.sol

    function getValuesInUsd(
        address creditor,
        address[] calldata assets,
        uint256[] calldata assetIds,
        uint256[] calldata assetAmounts
    ) public view returns (AssetValueAndRiskFactors[] memory valuesAndRiskFactors) {
        ...
        for (uint256 i; i < length; ++i) {
            (
                ...
            ) = IAssetModule(assetToAssetModule[assets[i]]).getValue(creditor,
            assets[i], assetIds[i], assetAmounts[i]);
>           if (valuesAndRiskFactors[i].assetValue < minUsdValue_)
valuesAndRiskFactors[i].assetValue = 0;
```

Note that derived assets are incorrectly priced regardless of the numeraire, but the example with the incorrectly priced numeraire illustrates the impact.

The below recommendation addresses the root cause and so all issues arising as a result of it will be mitigated.

Recommendation:

```
 ## Registry.sol

+    function getValuesInUsdNoMinUSD(
+        address creditor,
+        address[] calldata assets,
+        uint256[] calldata assetIds,
+        uint256[] calldata assetAmounts
+    ) public view returns (AssetValueAndRiskFactors[] memory valuesAndRiskFactors) {
+        uint256 length = assets.length;
+        valuesAndRiskFactors = new AssetValueAndRiskFactors[](length);
+
+        for (uint256 i; i < length; ++i) {
+            (
+                valuesAndRiskFactors[i].assetValue,
+                valuesAndRiskFactors[i].collateralFactor,
+                valuesAndRiskFactors[i].liquidationFactor
+            ) = IAssetModule(assetToAssetModule[assets[i]]).getValue(creditor,
assets[i], assetIds[i], assetAmounts[i]);
+        }
+    }
+


 ## AbstractDerivedAssetModule.sol

     /**
      * @notice Calculates the values per asset, denominated in a given Numeraire.
      * @param numeraire The contract address of the numeraire.

@@ -112,7 +112,7 @@ abstract contract DerivedAssetModule is AssetModule {
         }

         rateUnderlyingAssetsToUsd =
-            IRegistry(REGISTRY).getValuesInUsd(creditor, underlyingAssets,
underlyingAssetIds, amounts);
+            IRegistry(REGISTRY).getValuesInUsdNoMinUSD(creditor, underlyingAssets,
underlyingAssetIds, amounts);
     }
```

Arcadia: Addressed in PR 134.

Renascence: The recommendation has been implemented. In addition, in the instance where `AssetModule.getValue()` has been called with the `creditor` parameter, it is now called with the `address(0)` parameter (Link).

This is an additional improvement but not necessary since `rateNumeraireToUsd` no longer depends on `minUSDValue` and thereby the `creditor`.

[H-2] Reentrancy in Account initialization can be used to drain creditors funds

Context: Factory.sol, AccountV1.sol

Impact: Reentrancy in the `AccountV1` initialization flow can be used for theft of funds.

Description: In Arcadia, the entry point for account creation is `Factory.createAccount()`.

```
## Factory.sol

    function createAccount(uint256 salt, uint88 accountVersion, address creditor)
        external
        whenCreateNotPaused
        returns (address account)
    {
        accountVersion = accountVersion == 0 ? latestAccountVersion : accountVersion;

        if (accountVersion > latestAccountVersion) revert
        FactoryErrors.InvalidAccountVersion();
        if (accountVersionBlocked[accountVersion]) revert
        FactoryErrors.AccountVersionBlocked();

        // Hash tx.origin with the user provided salt to avoid front-running Account
        deployment with an identical salt.
        // We use tx.origin instead of msg.sender so that deployments through a third
        party contract are not vulnerable to front-running.
        account = address(
            new Proxy{ salt: keccak256(abi.encodePacked(salt, tx.origin)) }(
                versionInformation[accountVersion].implementation
            )
        );

        allAccounts.push(account);
        accountIndex[account] = allAccounts.length;

        _mint(msg.sender, allAccounts.length);

        IAccount(account).initialize(msg.sender,
        versionInformation[accountVersion].registry, creditor);

        emit AccountUpgraded(account, accountVersion);
    }
```

After the proxy has been deployed and an NFT has been minted, the function proceeds with initializing the proxy.

In this step, it's important to note that `locked = 1`, meaning the reentrancy guard is essentially **"unlocked"**, and there's an **external call** right after.

```
## AccountV1.sol

    function initialize(address owner_, address registry_, address creditor_)
    external {
        if (registry != address(0)) revert AccountErrors.AlreadyInitialized();
        if (registry_ == address(0)) revert AccountErrors.InvalidRegistry();
        owner = owner_;
>       locked = 1;
        registry = registry_;

>       if (creditor_ != address(0)) _openMarginAccount(creditor_);
    }
```

The final step is a call to open a margin account, which makes the external call to the `creditor`:

```
## AccountV1.sol

function _openMarginAccount(address creditor_) internal {
        (bool success, address numeraire_, address liquidator_, uint256
        fixedLiquidationCost_) =
            ICreditor(creditor_).openMarginAccount(ACCOUNT_VERSION);
        if (!success) revert AccountErrors.InvalidAccountVersion();

        fixedLiquidationCost = uint96(fixedLiquidationCost_);
        if (numeraire != numeraire_) _setNumeraire(numeraire_);

        emit MarginAccountChanged(creditor = creditor_, liquidator = liquidator_);
    }
```

The combination of the external call and the "unlocked" reentrancy guard opens the door for reentrancy, which can lead to direct theft of funds.

Assume the following scenario:

1. A malicious contract calls `Factory.createAccount()` and provides himself as the `creditor` the Account will be initialized with
2. After the initialization, the function flow proceeds with `AccountV1._openMarginAccount()`, which handles the execution back to the malicious contract:

```
## AccountV1.sol

function _openMarginAccount(address creditor_) internal {
>       (bool success, address numeraire_, address liquidator_, uint256
fixedLiquidationCost_) =
            ICreditor(creditor_).openMarginAccount(ACCOUNT_VERSION);
```

3. With the control back at the attacker contract, the attacker contract, which is also the `owner` of the Account, can deposit funds, open a margin account with a victim `creditor`, and take a loan from the victim `creditor`.
4. Once the attacker is done with the illicit actions, the execution flow continues in `AccountV1._openMarginAccount()`, and as a result, it sets the Account's `creditor` and the `liquidator` to the returned values of the malicious contract:

```
## AccountV1.sol

function _openMarginAccount(address creditor_) internal {
        emit MarginAccountChanged(creditor = creditor_, liquidator = liquidator_);
```

In the end, the attacker not only made the Account impossible to liquidate but also got complete control of both the loaned funds AND the deposited collateral.

Recommendation: Change the order of operations so the reentrancy guard is enabled during initialization:

```
## AccountV1.sol

contract AccountV1 is AccountStorageV1, IAccount {
        if (registry != address(0)) revert AccountErrors.AlreadyInitialized();
        if (registry_ == address(0)) revert AccountErrors.InvalidRegistry();
        owner = owner_;
-       locked = 1;
        registry = registry_;

        if (creditor_ != address(0)) _openMarginAccount(creditor_);
+       locked = 1;
    }
```

Arcadia: Addressed in PR 135.

Renascence: The recommendation has been implemented.

During initialization, the reentrancy guard is effectively enabled. So no other functions in `AccountV1` that have the `nonReentrant` modifier can be reentered, neither can the `AccountV1.initialize()` function itself (as a result of the `registry != address(0)` check).

At the time an Account is initialized, it does not have an existing affiliation with any creditor so we don't have to worry about affecting the existing creditor via e.g., a read-only reentrancy.

One action that can be performed on the Account is to transfer the ownership. But since the Account is just in the process of being set up, there is no concern about getting front-run before a new creditor is set.

The other two actions are setting the approved creditor and setting the asset manager. Neither of these actions can be leveraged in an attack.

[H-3] AccountV1.setApprovedCreditor() doesnt trigger a transfer delay and can be used to steal funds through AccountV1.flashActionByCreditor()

Context: Factory.sol, AccountV1.sol

Impact: A malicious `owner` can set an `approvedCreditor` before the account transfer and then steal the funds after the transfer via `AccountV1.flashActionByCreditor()`.

Description: In Arcadia, Accounts are created through the `Factory` contract by minting an NFT.

```
## Factory.sol

    function createAccount(uint256 salt, uint88 accountVersion, address creditor)
        external
        whenCreateNotPaused
        returns (address account)
    {
        ...

>       _mint(msg.sender, allAccounts.length);

        IAccount(account).initialize(msg.sender,
        versionInformation[accountVersion].registry, creditor);

        emit AccountUpgraded(account, accountVersion);
    }
```

After creation, Accounts are transferable.

```
## Factory.sol

    function safeTransferFrom(address from, address to, address account) public {
        uint256 id = accountIndex[account];
        IAccount(allAccounts[id - 1]).transferOwnership(to);
        super.safeTransferFrom(from, to, id);
    }
```

To open a margin account, the newly created Account has to be associated with a `creditor`.

```
## AccountV1.sol

    function openMarginAccount(address newCreditor)
        external
        onlyOwner
        nonReentrant
        notDuringAuction
        updateActionTimestamp
    {
        ...
        // Remove the exposures of the Account for the old Creditor.
        if (oldCreditor != address(0)) {
            IRegistry(registry).batchProcessWithdrawal(oldCreditor, assetAddresses,
            assetIds, assetAmounts);
        }

        // Open margin account for the new Creditor.
>       _openMarginAccount(newCreditor);
```

There's also the possibility of setting an `approvedCreditor`. An `approvedCreditor` is a `creditor` for which no margin account is immediately opened.

Both types of creditors can be used to call `AccountV1.flashActionByCreditor()`.

```
## AccounV1.sol

    function flashActionByCreditor(address actionTarget, bytes calldata actionData)
        external
        nonReentrant
        notDuringAuction
        updateActionTimestamp
        returns (uint256 accountVersion)
    {
        // Cache the current Creditor.
        address currentCreditor = creditor;

        // The caller has to be or the Creditor of the Account, or an approved
        Creditor.
        if (msg.sender != currentCreditor && msg.sender != approvedCreditor) revert
        AccountErrors.OnlyCreditor();
```

The vulnerability here stems from the fact that when calling `AccountV1.setApprovedCreditor()`, the `updateActionTimestamp` modifier is not triggered; hence a cool-down period is not started.

This can be problematic because a malicious `owner` can use it to set an `approvedCreditor` before the account transfer and then use the `approvedCreditor` to steal funds from the Account he has just transferred.

Assume the following scenario:

1. Amelie is going to buy Noemi's Account
2. Noemi calls `Factory.safeTransferFrom()` to transfer the Account.
3. However, right before transferring it, she also calls `AccountV1.setApprovedCreditor(0xNoemi)`
4. Even though Amelie owns the Account, Noemi is her `approvedCreditor`
5. Noemi utilizes `AccountV1.flashActionByCreditor()` to drain all the funds in the just-transferred Account.

Recommendation: Apply the `updateActionTimestamp` modifier to `AccountV1.setApprovedCreditor()` to trigger the cool-down period.

```
 ## AccountV1.sol

 contract AccountV1 is AccountStorageV1, IAccount {
-    function setApprovedCreditor(address creditor_) external onlyOwner {
+    function setApprovedCreditor(address creditor_) external onlyOwner
updateActionTimestamp {
         approvedCreditor = creditor_;
     }
```

Arcadia: Addressed in PR 132.

Renascence: The recommendation has been implemented.

[H-4] Reentrancy during liquidation corrupts the LendingPool accounting completely

Context: AccountV1.sol, LendingPool.sol, Liquidator.sol

Impact: Reentrancy in the liquidation process will corrupt the debt accounting by inflating the `totalRealisedLiquidity` variable and `realisedLiquidityOf` mapping.

The ongoing auctions count variable `auctionsInProgress` can get decremented more than once for the same auction, locking the Junior Tranche forever.

`LendingPool.skim()` can be unavailable while there are no ongoing auctions and available during an ongoing auction.

Description: To participate in a liquidation auction, users must call `Liquidator.bid()` with the asset amounts they wish to buy in return for repaying some of the debt.

During `Liquidator.bid()`, the debt repayment is handled through `LendingPool.auctionRepay()`, after which `IAccount.auctionBid()` is responsible for transferring the asked amounts of assets to the bidding user.

The auction terminates if all of the debt is paid back during `LendingPool.auctionRepay()`.

If the Account is back in an overcollateralized "healthy" state or all of the collateral is sold out or `block.timestamp > auctionInformation_.cutoffTimeStamp`, the user could also pass `true` in the `endAuction_` variable and manually terminate ("settle") the auction.

An important detail is a separate `Liquidator.endAuction()` function that can be used to settle an auction without purchasing collateral.

```
## Liquidator.sol

    function bid(address account, uint256[] memory askedAssetAmounts, bool
    endAuction_) external {
        AuctionInformation storage auctionInformation_ = auctionInformation[account];
        if (!auctionInformation_.inAuction) revert LiquidatorErrors.NotForSale();
        ...
         bool earlyTerminate =
>        ILendingPool(auctionInformation_.creditor).auctionRepay(startDebt, price,
account, msg.sender);

>        IAccount(account).auctionBid(
            auctionInformation_.assetAddresses, auctionInformation_.assetIds,
            askedAssetAmounts, msg.sender);

          if (earlyTerminate) {
            _endAuction(account);
           }

          else if (endAuction_) {
            if (_settleAuction(account, auctionInformation_)) _endAuction(account);
          }
```

```
## Liquidator.sol

    function endAuction(address account) external {
        AuctionInformation storage auctionInformation_ = auctionInformation[account];

        // Check if the Account is being auctioned.
        if (!auctionInformation_.inAuction) revert LiquidatorErrors.NotForSale();

        if (!_settleAuction(account, auctionInformation_)) revert
        LiquidatorErrors.EndAuctionFailed();

        _endAuction(account);
    }
```

Back in `Liquidator.bid()`, the `AccountV1.auctionBid()` function does an internal call to `AccountV1._withdraw()` where asset withdrawals are accounted for based on their type and finally transferred to the bidding user.

It's important to note that ERC777, ERC721, and ERC1155 assets can now be used to trigger an "`onReceived()`" hook if the bidding user happens to be a smart contract.

Moreover, the `AccountV1.endAuction()` function within `Liquidator._endAuction()` doesn't have a `nonReentrant` guard. This opens up the possibility for a reentrancy attack that starts with a call to `Liquidator.bid()` and during the `onReceived()` hook a call is made to `Liquidator.endAuction()`.

```
## AccountV1.sol

    function _withdraw(
        address[] memory assetAddresses,
        uint256[] memory assetIds,
        uint256[] memory assetAmounts,
        address to
    ) internal {
        // Reverts in Registry if input is invalid.
        uint256[] memory assetTypes =
            IRegistry(registry).batchProcessWithdrawal(creditor, assetAddresses,
            assetIds, assetAmounts);

        for (uint256 i; i < assetAddresses.length; ++i) {
            if (assetAmounts[i] == 0) {
                // Skip if amount is 0 to prevent transferring 0 balances.
                continue;
            }

            if (assetTypes[i] == 0) {
                if (assetIds[i] != 0) revert AccountErrors.InvalidERC20Id();
>                _withdrawERC20(to, assetAddresses[i], assetAmounts[i]);
            } else if (assetTypes[i] == 1) {
                if (assetAmounts[i] != 1) revert AccountErrors.InvalidERC721Amount();
>                _withdrawERC721(to, assetAddresses[i], assetIds[i]);
            } else if (assetTypes[i] == 2) {
>                _withdrawERC1155(to, assetAddresses[i], assetIds[i],
assetAmounts[i]);
            } else {
                revert AccountErrors.UnknownAssetType();
            }
        }
    }
```

Here is a thorough breakdown of how the exploit would work and its consequences.

1. Assume a Liquidation process has started for some `Account`, referred to as X → `Liquidator.liquidateAccount(address(X))`. Let's also assume that the Account holds a single ERC721 asset.
2. Amelie, a malicious actor, notices the liquidation process for X has started and deploys the following `MaliciousReceiver` contract that has implemented an `IERC721TokenReceiver.onERC721Received()` hook function.

```
contract MaliciousReceiver is ERC721TokenReceiver{
    bool flag;
    address liq;
    address acc;
    address lending_pool;
    address nft_addr;
    constructor(address pool, address token, address _nft_addr) {

        ERC20(token).approve(pool, type(uint128).max);
        lending_pool = pool;
        nft_addr = _nft_addr;
    }

    function bid(address liquidator, address account) external {

        liq = liquidator;
        acc = account;
        uint[] memory amt = new uint[](1);
        amt[0] = 1;
        Liquidator(liquidator).bid(account, amt, false);
    }

    function onERC721Received( address operator, address from, uint256 tokenId, bytes
    calldata data) external override returns (bytes4) {

        if(flag == false) {
            flag = true;

>           Liquidator(liq).endAuction(acc);
            return ERC721TokenReceiver.onERC721Received.selector;
```

3. Amelie now transfers enough funds to `MaliciousReceiver` to repay the debt of Account X and receive the ERC721 asset.
4. Amelie calls `MaliciousReceiver.bid()` with the address of Account X, which calls `Liquidator.bid()`
5. In `Liquidator.bid()`, the call to `LendingPool.auctionRepay()` will pay back the debt of Account X, and the important internal call to `LendingPool._settleLiquidationHappyFlow()` will be made.

In this internal function, `totalRealisedLiquidty` state variable is updated by adding the Liquidation Incentives (`terminationReward` and `liquidationPenalty`) and any surplus of repaid debt.

Moreover, the terminator, which in this case is `MaliciousReceiver`, will receive the `terminationReward` added in `realisedLiquidityOf[terminator]`.

Lastly, the internal call to `LendingPool._endLiquidation()` is made where `auctionsInProgress` is decremented.

```
## LendingPool.sol

  function _settleLiquidationHappyFlow(address account, uint256 startDebt, address
  terminator, uint256 surplus)
        internal
    {
        (uint256 initiationReward, uint256 terminationReward, uint256
        liquidationPenalty) = _calculateRewards(startDebt);

        _syncLiquidationFeeToLiquidityProviders(liquidationPenalty);

>       totalRealisedLiquidity = uint128(totalRealisedLiquidity + terminationReward +
liquidationPenalty + surplus);

        unchecked {
            if (surplus > 0) realisedLiquidityOf[IAccount(account).owner()] +=
            surplus;
            // Pay out the "terminationReward" to the "terminator".
>           realisedLiquidityOf[terminator] += terminationReward;
        }

>       _endLiquidation();

        emit AuctionFinished(
            account, address(this), startDebt, initiationReward, terminationReward,
            liquidationPenalty, 0, surplus
        );
    }
```

```
## LendingPool.sol

  function _endLiquidation() internal {
        // Decrement the number of auctions in progress.
        unchecked {
>            --auctionsInProgress;
        }

        // Hook to the most junior Tranche.
        if (auctionsInProgress == 0 && tranches.length > 0) {
            ITranche(tranches[tranches.length - 1]).setAuctionInProgress(false);
        }

        // Event emitted by Liquidator.
    }
```

6. After `LendingPool.repayAuction()` concludes, `Liquidator.bid()` continues with a call to `IAccount.auctionBid()` which will send the ERC721 asset to `MaliciousContract`, triggering `MaliciousContract.onERC721Received()`
7. The hook will then call `Liquidator.endAuction()`, which will not revert since `auctionInformation` hasn't been deleted yet by `Liquidator.bid()` through which the reentrancy happens.

The internal call to `Liquidator_settleAuction()` will then call `LendingPool.settleLiquidationHappyFlow()`, which will finally call the internal `LendingPool._settleLiquidationHappyFlow()`.

This means the auction of Account X will be settled for a second time with double accounting of `totalRealisedLiquidty` and `realisedLiquidityOf[terminator]`.

Additionally, `auctionsInProgress` will get decremented once more and could underflow to `type(uint16).max`. 8. The internal function `Liquidator._endAuction()` will be called as part of `Liquidator.endAuction()` and won't revert since `AccountV1.endAuction()` doesn't have a `nonReentrant` modifier. Lastly, execution will resume in `bid()` where `Liquidator._endAuction()` will be called once more.

Recommendation

Add `nonReentrant` modifiers to `AccountV1.auctionBid`, `AccountV1.endAuction()`, and `AccountV1.auctionBoughtIn()`.

```
 ## AccountV1.sol

@@ -548,7 +548,7 @@ contract AccountV1 is AccountStorageV1, IAccount {
         uint256[] memory assetIds,
         uint256[] memory assetAmounts,
         address bidder
-    ) external onlyLiquidator {
+    ) external onlyLiquidator nonReentrant {
         _withdraw(assetAddresses, assetIds, assetAmounts, bidder);
     }

@@ -558,14 +558,14 @@ contract AccountV1 is AccountStorageV1, IAccount {
      * @dev When an auction is not successful, the Account is considered "Bought In":
      * The whole Account including any remaining assets are transferred to a certain
      recipient address, set by the Creditor.
      */
-    function auctionBoughtIn(address recipient) external onlyLiquidator {
+    function auctionBoughtIn(address recipient) external onlyLiquidator nonReentrant
{
         _transferOwnership(recipient);
     }

     /**
      * @notice Sets the "inAuction" flag to false when an auction ends.
      */
-    function endAuction() external onlyLiquidator {
+    function endAuction() external onlyLiquidator nonReentrant {
         inAuction = false;
     }
```

Arcadia: Addressed in PR 135 and PR 136.

Renascence: The specific attack vector presented in this finding has been fixed by applying the `nonReentrant` modifier to `AccountV1.auctionBid()` and `AccountV1.endAuction()`.

All paths to interact with an auction call either `AccountV1.auctionBid()` or `AccountV1.endAuction()` and so reentering from the token transfer callback doesn't allow one to interact with the auction again.

To reduce the attack surface of the creditor's `accountRecipient`, the `nonReentrant` modifier has been applied to `AccountV1.auctionBoughtIn()` as well.

Note that the fix for [H-8] provides an additional layer of security by applying reentrancy guards to

the `Liquidator` itself.

[H-5] Liquidation of unhealthy accounts can be prevented at low cost

Context: Liquidator.sol

Impact: Liquidation of unhealthy accounts can be halted even though they should still be liquidatable.

Description: `Liquidator.bid()` is the entry point for bidding on liquidation auctions. Since this finding is primarily concerned with halting the liquidation, the other part of the function is omitted for simplicity.

It's important to acknowledge that setting `endAuction_ = true` would call `Liquidator._endAuction()` if `Liquidator._settleAuction()` returns true.

```
## Liquidator.sol

  function bid(address account, uint256[] memory askedAssetAmounts, bool endAuction_)
  external {
        ...
        if (earlyTerminate) {
            _endAuction(account);
        }
        else if (endAuction_) {
>           if (_settleAuction(account, auctionInformation_)) _endAuction(account);
        }
    }
```

Observing `Liquidator._settleAuction()`, we can see that it first sets `inAuction` to false and then checks the three conditions below.

If none of the conditions are met, the function returns false, meaning `Liquidator_endAuction()` from the previous step won't be called.

```
## Liquidator.sol

 function _settleAuction(address account, AuctionInformation storage
 auctionInformation_)
        internal
        returns (bool success)
    {
        // Stop the auction.
>       auctionInformation_.inAuction = false;

        // Cache variables.
        uint256 startDebt = auctionInformation_.startDebt;
        address creditor = auctionInformation_.creditor;

        uint256 collateralValue = IAccount(account).getCollateralValue();
        uint256 usedMargin = IAccount(account).getUsedMargin();

        if (collateralValue >= usedMargin) {
            ILendingPool(creditor).settleLiquidationHappyFlow(account, startDebt,
            msg.sender);
        } else if (collateralValue == 0) {
            ILendingPool(creditor).settleLiquidationUnhappyFlow(account, startDebt,
            msg.sender);
        } else if (block.timestamp > auctionInformation_.cutoffTimeStamp) {
            ILendingPool(creditor).settleLiquidationUnhappyFlow(account, startDebt,
            msg.sender);
            IAccount(account).auctionBoughtIn(creditorToAccountRecipient[creditor]);
        } else {
            // None of the conditions to end the auction are met.
>           return false;
        }

        return true;
    }
```

However, the fact that `auctionInformation_.inAuction` is updated regardless of whether the auction is settled or not leads to a critical vulnerability where the liquidation auction of an unhealthy account can be ended for a marginal cost.

Assume the following scenario:

1. Alice's account is unhealthy
2. She makes a call to `Liquidation.bid()` with negligible arbitrary asset amounts, but passes `endAuction=true`
3. `Liquidator._settleAuction()` sets `inAuction=false`
4. The auction is stopped even though Alice is still liquidatable

Here's a coded, runnable POC that can be used as a reference:

```
function testFuzz_Success_bid_partially(address bidder, uint112 amountLoaned) public {
        // Given: The account auction is initiated
        vm.assume(bidder != address(0));
        vm.assume(amountLoaned > 12);
        vm.assume(amountLoaned <= (type(uint112).max / 300) * 100);
        initiateLiquidation(amountLoaned);
        bool endAuction = true;

        uint256[] memory originalAssetAmounts =
        liquidator.getAuctionAssetAmounts(address(proxyAccount));
        uint256 originalAmount = originalAssetAmounts[0];

        // And: Bidder has enough funds and approved the lending pool for repay
        uint256[] memory bidAssetAmounts = new uint256[](1);
        uint256 bidAssetAmount = originalAmount / 50;
        bidAssetAmounts[0] = bidAssetAmount;
        deal(address(mockERC20.stable1), bidder, type(uint128).max);
        vm.startPrank(bidder);
        mockERC20.stable1.approve(address(pool), type(uint256).max);

        // When: Bidder bids for the asset
        liquidator.bid(address(proxyAccount), bidAssetAmounts, endAuction);
        vm.stopPrank();

        // Then: The bidder should have the asset, and left assets should be
        diminished
        //        uint256 totalBids =
        liquidator.getAuctionTotalBids(address(proxyAccount));
        //        uint256 askPrice =
        liquidator.calculateAskPrice(address(proxyAccount), bidAssetAmounts, new
        uint256[](1));
        //        assertEq(totalBids, askPrice);

        // And: Auction is still going on since the bidder did not choose the end the
        endAuction
        bool inAuction = liquidator.getInAuction(address(proxyAccount));
        assertEq(inAuction, false);
        bool liquidatable = proxyAccount.isAccountLiquidatable();
        assertEq(liquidatable, true);
    }
```

Recommendation:

```
 ## Liquidator.sol

    function _settleAuction(address account, AuctionInformation storage
    auctionInformation_)
        internal
        returns (bool success)
    {
-
-        auctionInformation_.inAuction = false;
-
        // Cache variables.
        uint256 startDebt = auctionInformation_.startDebt;
        address creditor = auctionInformation_.creditor;
        ...
```

Arcadia: Addressed in PR 97.

Renascence: The recommendation has been implemented.

[H-6] Misjudgment regarding reentrancy risks associated with ERC777 tokens can lead to direct theft of funds

Context: LendingPool.sol, AccountV1.sol, Liquidator.sol

Impact: ERC777 tokens enable multiple reentrancy attacks. The vulnerability stems from the fact that ERC777 tokens, unlike ERC1155 and ERC721 tokens, allow the `from` address of a transfer to get a callback, which introduces an additional attack surface that has been unaccounted for.

Description: In the pre-audit questionnaire, the client states that the protocol should be compatible with ERC777 tokens. This has been confirmed during consultation with the client.

However, that's not true, ERC777 tokens have a `tokensToSend` hook that allows the `from` address of a transfer to get a callback.

Here's a reference to an ERC777 implementation:

https://github.com/Switcheo/switcheo-eth/blob/master/contracts/lib/token/ERC777/ERC777.sol

```
    function transferFrom(address holder, address recipient, uint256 amount) external
    returns (bool) {
        require(recipient != address(0), "ERC777: transfer to the zero address");
        require(holder != address(0), "ERC777: transfer from the zero address");

        address spender = msg.sender;

>       _callTokensToSend(spender, holder, recipient, amount, "", "");

        _move(spender, holder, recipient, amount, "", "");
        _approve(holder, spender, _allowances[holder][spender].sub(amount));

        _callTokensReceived(spender, holder, recipient, amount, "", "", false);

        return true;
    }
```

There are three separate vulnerabilities arising from this misconception about ERC777 tokens in the codebase, so we'll go through them one by one:

Let's take a look at `LendingPool.donateToTranche()`.

```
## LendingPool.sol

    function donateToTranche(uint256 trancheIndex, uint256 assets) external
    whenDepositNotPaused processInterests {
        if (assets == 0) revert LendingPoolErrors.ZeroAmount();

        address tranche = tranches[trancheIndex];
        // Mitigate share manipulation, where first Liquidity Provider mints just 1
        share.
        // See https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3706 for
        more information.
        if (ERC4626(tranche).totalSupply() < 10 ** decimals) revert
        LendingPoolErrors.InsufficientShares();

>       asset.safeTransferFrom(msg.sender, address(this), assets);

        unchecked {
            realisedLiquidityOf[tranche] += assets; //[  $  (    °     °  )  $  ]
            totalRealisedLiquidity = SafeCastLib.safeCastTo128(assets +
            totalRealisedLiquidity);
        }
    }
```

In this case, an attacker can get the callback after the checks but before the transfer.

Consequently, he can bypass the check, burn shares such that only `1 share` is left, and then perform an inflation attack through donation.

The next vulnerability is in `LendingPool.auctionRepay()`, which allows to reenter the liquidation process because the `Liquidator` lacks reentrancy guards.

```
## LendingPool.sol

  function auctionRepay(uint256 startDebt, uint256 amount, address account, address
  bidder)
        external
        whenLiquidationNotPaused
        onlyLiquidator
        processInterests
        returns (bool earlyTerminate)
    {
        // Need to transfer before burning debt or ERC777s could reenter.
        // Address(this) is trusted -> no risk on re-entrancy attack after transfer.
>       asset.safeTransferFrom(bidder, address(this), amount);
```

A detailed description of the reentrancy process is provided in another issue that deals with a different reentrancy vulnerability in the `Liquidator`.

The last vulnerability is in `LendingPool.flashAction()`.

```
## LendingPool.sol

    function flashAction(
        uint256 amountBorrowed,
        address account,
        address actionTarget,
        bytes calldata actionData,
        bytes3 referrer
    ) external whenBorrowNotPaused processInterests {
        ...


        // Send Borrowed funds to the actionTarget.
>       asset.safeTransfer(actionTarget, amountBorrowed);
```

Here, the attacker can bypass the transfer ownership delay in `AccountV1` by transferring the ownership in the callback.

Then they can still continue withdrawing funds from the Account within the `flashAction`.

Recommendation: The mitigation for the inflation attack:

```
 ## LendingPool.sol

@@ -374,13 +374,13 @@ contract LendingPool is LendingPoolGuardian, Creditor,
DebtToken, ILendingPool {
     function donateToTranche(uint256 trancheIndex, uint256 assets) external
     whenDepositNotPaused processInterests {
         if (assets == 0) revert LendingPoolErrors.ZeroAmount();

+        asset.safeTransferFrom(msg.sender, address(this), assets);
+
         address tranche = tranches[trancheIndex];
         // Mitigate share manipulation, where first Liquidity Provider mints just 1
         share.
         // See https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3706 for
         more information.
         if (ERC4626(tranche).totalSupply() < 10 ** decimals) revert
         LendingPoolErrors.InsufficientShares();

-        asset.safeTransferFrom(msg.sender, address(this), assets);
-
         unchecked {
             realisedLiquidityOf[tranche] += assets; //[  $  (    °     °  )  $  ]
             totalRealisedLiquidity = SafeCastLib.safeCastTo128(assets +
             totalRealisedLiquidity);
```

The mitigation for the reentrancy in the liquidation flow:

```
 ## Liquidator.sol

@@ -34,7 +34,7 @@ contract Liquidator is Owned, ILiquidator {
```

```
     /* ////////////////////////////////////////////////////////////////
                                  STORAGE
     //////////////////////////////////////////////////////////////// */
-
+    uint8 internal locked;
     // The base of the auction price curve (decreasing power function).
     // Determines in what time the auction price halves, 18 decimals precision.
     uint64 internal base;
@@ -87,6 +87,13 @@ contract Liquidator is Owned, ILiquidator {
         uint64 base, uint32 cutoffTime, uint16 startPriceMultiplier, uint16
         minPriceMultiplier
     );

+    modifier nonReentrant() {
+        if (locked != 1) revert AccountErrors.NoReentry();
+        locked = 2;
+        _;
+        locked = 1;
+    }
+
     /* ////////////////////////////////////////////////////////////////
                                  CONSTRUCTOR
     //////////////////////////////////////////////////////////////// */
@@ -107,6 +114,8 @@ contract Liquidator is Owned, ILiquidator {
         // 60%.
         minPriceMultiplier = 6000;

+        locked = 1;
+
         emit AuctionCurveParametersSet(999_807_477_651_317_446, 14_400, 15_000,
         6000);
     }

@@ -191,7 +200,7 @@ contract Liquidator is Owned, ILiquidator {
      * @notice Initiate the liquidation of an Account.
      * @param account The contract address of the Account to be liquidated.
      */
-    function liquidateAccount(address account) external {
+    function liquidateAccount(address account) external nonReentrant {
         if (!IFactory(ACCOUNT_FACTORY).isAccount(account)) revert
         LiquidatorErrors.IsNotAnAccount();

         AuctionInformation storage auctionInformation_ = auctionInformation[account];
@@ -275,7 +284,7 @@ contract Liquidator is Owned, ILiquidator {
      * @dev The bidder is not obliged to set endAuction to True if the Account is
      healthy after the bid,
      * but they are incentivised to do so by earning an additional
      "auctionTerminationReward".
      */
-    function bid(address account, uint256[] memory askedAssetAmounts, bool
endAuction_) external {
+    function bid(address account, uint256[] memory askedAssetAmounts, bool
endAuction_) external nonReentrant {
         AuctionInformation storage auctionInformation_ = auctionInformation[account];
         if (!auctionInformation_.inAuction) revert LiquidatorErrors.NotForSale();

@@ -396,7 +405,7 @@ contract Liquidator is Owned, ILiquidator {
      * @notice Ends an auction and settles the liquidation.
      * @param account The contract address of the Account in liquidation.
```

```
     */
-    function endAuction(address account) external {
+    function endAuction(address account) external nonReentrant {
         AuctionInformation storage auctionInformation_ = auctionInformation[account];

         // Check if the Account is being auctioned.
```

The mitigation for the bypass of the ownership transfer delay:

```
 ## LendingPool.sol

@@ -563,6 +563,8 @@ contract LendingPool is LendingPoolGuardian, Creditor, DebtToken,
ILendingPool {
         address accountOwner = IFactory(ACCOUNT_FACTORY).ownerOfAccount(account);
         if (accountOwner == address(0)) revert LendingPoolErrors.IsNotAnAccount();

+        IAccount(account).updateActionTimestampCreditor();
+
         uint256 amountBorrowedWithFee = amountBorrowed +
         amountBorrowed.mulDivUp(originationFee, ONE_4);

         // Check allowances to take debt.
```

```
 ## AccountV1.sol

@@ -234,6 +234,8 @@ contract AccountV1 is AccountStorageV1, IAccount {
         }
     }

+    function updateActionTimestampCreditor() onlyCreditor updateActionTimestamp {}
+
     /**
      * @notice Finalizes the Upgrade to a new Account version on the new
      implementation Contract.
      * @param oldImplementation The old contract address of the Account
      implementation.
```

Arcadia: Addressed in PR 143 and PR 104.

Renascence: All recommendations for the three different attack vectors have been implemented.

Every instance of an ERC20 transfer has been checked whether there is a reentrancy risk if ERC777 tokens are used. No further issues have been found.

4.2 Medium Risk

[M-1] Under certain conditions, direct theft of funds is possible during liquidation

Context: AccountV1.sol, Liquidator.sol

Impact: The absence of the `notDuringAuction` modifier in the `AccountV1.deposit()` function allows a scenario where a malicious actor can take advantage of the liquidation process. Specifically, a liquidator can bid for assets with a stale share amount that does not consider newly deposited collateral. This results in the liquidator acquiring additional assets for free.

Description: Contrary to withdrawing assets, depositing new collateral into an Account during liquidation is possible because of a missing `notDuringAuction` modifier.

```
## AccountV1.sol

    function deposit(address[] calldata assetAddresses, uint256[] calldata assetIds,
    uint256[] calldata assetAmounts)
        external
        onlyOwner
        nonReentrant
    {
        // No need to check that all arrays have equal length, this check will be done
        in the Registry.
>       _deposit(assetAddresses, assetIds, assetAmounts, msg.sender);
    }
```

Assume that an Account holds almost exclusively USDT and a small amount of USDC at the start of a liquidation where the value of USDC doesn't meet the `minUSDValue` threshold.

```
## Registry.sol

  function getValuesInUsd(
        address creditor,
        address[] calldata assets,
        uint256[] calldata assetIds,
        uint256[] calldata assetAmounts
    ) public view returns (AssetValueAndRiskFactors[] memory valuesAndRiskFactors) {
        uint256 length = assets.length;
        valuesAndRiskFactors = new AssetValueAndRiskFactors[](length);

>       uint256 minUsdValue_ = minUsdValue[creditor];
        for (uint256 i; i < length; ++i) {
             ...
>            if (valuesAndRiskFactors[i].assetValue < minUsdValue_)
valuesAndRiskFactors[i].assetValue = 0;
        }
    }
```

During the calculation of the shares for the collateral that will be liquidated, the asset share for USDC will be set to 0.

```
## Liquidator.sol

function _getAssetShares(AssetValueAndRiskFactors[] memory assetValues)
    internal
    pure
    returns (uint32[] memory assetShares)
{
    ...
    for (uint256 i; i < length; ++i) {
        unchecked {
>           assetShares[i] = uint32(assetValues[i].assetValue * ONE_4 / totalValue);
        }
    }
}
```

However, if the Account owner now decides to deposit additional collateral of 1000 USDC, a malicious liquidator can observe the deposit and immediately call `Liquidator.bid(, uint256[] memory askedAssetAmounts, )` with amounts of 1 USDT and 1000 USDC. Now, even though the Account owner just deposited more collateral, such a bid will use the already stored auction information where the shares for USDC are 0. As a consequence, the overall bid price calculation won't be inclusive of the 1000 USDC and the malicious bidder will receive the collateral for free.

Note: The 1 USDT is required because the function would otherwise revert.

```
## Liquidator.sol

function bid(address account, uint256[] memory askedAssetAmounts, bool endAuction_)
external {
        AuctionInformation storage auctionInformation_ = auctionInformation[account];
        if (!auctionInformation_.inAuction) revert LiquidatorErrors.NotForSale();

        // Calculate the current auction price of the assets being bought.
        uint256 totalShare = _calculateTotalShare(auctionInformation_,
        askedAssetAmounts);
```

```
## Liquidator.sol

    function _calculateTotalShare(AuctionInformation storage auctionInformation_,
    uint256[] memory askedAssetAmounts)
        internal
        view
        returns (uint256 totalShare)
    {
        uint256[] memory assetAmounts = auctionInformation_.assetAmounts;
>       uint32[] memory assetShares = auctionInformation_.assetShares;
        if (assetAmounts.length != askedAssetAmounts.length) {
            revert LiquidatorErrors.InvalidBid();
        }

        for (uint256 i; i < askedAssetAmounts.length; ++i) {
            unchecked {
>               totalShare += askedAssetAmounts[i] * assetShares[i] / assetAmounts[i];
            }
        }
```

Recommendation: Apply the `notDuringAuction` modifier to the deposit function.

```
 ## AccountV1.sol

    function deposit(address[] calldata assetAddresses, uint256[] calldata assetIds,
    uint256[] calldata assetAmounts)
        external
        onlyOwner
        nonReentrant
+       notDuringAuction
    {
        // No need to check that all arrays have equal length, this check will be done
        in the Registry.
        _deposit(assetAddresses, assetIds, assetAmounts, msg.sender);
    }
```

Notice that the recommendation also solves an edge case where liquidations are unable to handle account deposits:

Specifically, suppose new assets are deposited while the liquidation is ongoing. In that case, it would be impossible to reach the `collateralValue == 0` condition because if new assets are deposited they cannot get liquidated in the ongoing liquidation. Consequently, liquidations might be delayed until the cutoff time is reached.

Arcadia: Addressed in PR 131.

Renascence: The recommendation has been implemented.

[M-2] Zero account value renders liquidations impossible

Context: Liquidator.sol

Impact: The liquidation process fails to handle edge cases where collateral values drop below a specified minimum (`minUSDValue`). This leads to a division by zero error during the `assetShares` cal-

culation, rendering liquidations impossible and bad debt remaining unresolved.

Description: When triggering a liquidation, the function computes the `assetShares` by calculating the relative value of each asset in relation to the `totalValue` of the Account.

```
## Liquidator.sol

   function liquidateAccount(address account) external {
        ...
>       auctionInformation_.assetShares = _getAssetShares(assetValues);
    }
```

```
## Liquidator.sol

function _getAssetShares(AssetValueAndRiskFactors[] memory assetValues)
        internal
        pure
        returns (uint32[] memory assetShares)
    {
        uint256 length = assetValues.length;
        uint256 totalValue;
        for (uint256 i; i < length; ++i) {
            unchecked {
>               totalValue += assetValues[i].assetValue;
            }
        }
        assetShares = new uint32[](length);
        for (uint256 i; i < length; ++i) {
            unchecked {
                // The asset shares are calculated relative to the total value of the
                Account.
                // "assetValue" is a uint256 in Numeraire units, will never overflow.
>               assetShares[i] = uint32(assetValues[i].assetValue * ONE_4 /
totalValue);
            }
        }
    }
```

However, the collateral can be determined to have zero value when it drops below `minUSDValue`.

```
## Registry.sol

  function getValuesInUsd(
        address creditor,
        address[] calldata assets,
        uint256[] calldata assetIds,
        uint256[] calldata assetAmounts
    ) public view returns (AssetValueAndRiskFactors[] memory valuesAndRiskFactors) {
        uint256 length = assets.length;
        valuesAndRiskFactors = new AssetValueAndRiskFactors[](length);

>       uint256 minUsdValue_ = minUsdValue[creditor];
        for (uint256 i; i < length; ++i) {
             ...
>           if (valuesAndRiskFactors[i].assetValue < minUsdValue_)
valuesAndRiskFactors[i].assetValue = 0;
        }
    }
```

Consequently, this would make the call to `Liquidator.liquidateAccount()` revert because it would require a division by 0.

```
## Liquidator.sol

function _getAssetShares(AssetValueAndRiskFactors[] memory assetValues)
        internal
        pure
        returns (uint32[] memory assetShares)
    {
          ...
>         assetShares[i] = uint32(assetValues[i].assetValue * ONE_4 / totalValue);
    }
```

The result is that liquidation is impossible, and the bad debt isn't removed.

Recommendation:

```
 ## Liquidator.sol

    function _getAssetShares(AssetValueAndRiskFactors[] memory assetValues)
        internal
        pure
        returns (uint32[] memory assetShares)
    {
        uint256 length = assetValues.length;
        uint256 totalValue;
        for (uint256 i; i < length; ++i) {
            unchecked {
                totalValue += assetValues[i].assetValue;
            }
        }
        assetShares = new uint32[](length);
+
+       if (totalValue == 0) {return assetShares;}
+
        for (uint256 i; i < length; ++i) {
            unchecked {
                // The asset shares are calculated relative to the total value of the
                Account.
                // "assetValue" is a uint256 in Numeraire units, will never overflow.
                assetShares[i] = uint32(assetValues[i].assetValue * ONE_4 /
                totalValue);
            }
        }
    }
```

Arcadia: Addressed in PR 98.

Renascence: The recommendation has been implemented.

[M-3] UniswapV3AssetModule exposure can be manipulated by withdrawing zero amounts

Context: UniswapV3AssetModule.sol

Impact: Exposure limits for Uniswap V3 positions can be bypassed. The exposure limits are a defense mechanism against price manipulation attacks and so bypassing them would allow an attacker to deposit large amounts of inflated assets to then steal other assets.

Description: The `AbstractDerivedAssetModule` keeps track of the exposure of each asset.

```
## AbstractDerivedAssetModule.sol

    // Struct with the exposures of a specific asset for a specific Creditor.
    struct ExposuresPerAsset {
        // The amount of exposure of the Creditor to the asset at the last
        interaction.
        uint112 lastExposureAsset;
        // The exposure in USD of the Creditor to the asset at the last interaction,
        18 decimals precision.
        uint112 lastUsdExposureAsset;
    }
```

There are checks for the exposure to block deposits that would over-expose a `Creditor` to a particular asset or protocol.

For Uniswap V3, every token is a unique asset. Hence, for every deposit, the token must be added to the Asset Module.

Note how `assetToLiquidity[assetId] = liquidity;` maps the `assetId` to the `liquidity`.

```
## UniswapV3AssetModule.sol

    function _addAsset(uint256 assetId) internal {
        if (assetId > type(uint96).max) revert InvalidId();

        (,, address token0, address token1,,,, uint128 liquidity,,,,) =
        NON_FUNGIBLE_POSITION_MANAGER.positions(assetId);

        // No need to explicitly check if token0 and token1 are allowed, _addAsset()
        is only called in the
        // deposit functions and there any deposit of non-allowed Underlying Assets
        will revert.
        if (liquidity == 0) revert ZeroLiquidity();

        // The liquidity of the Liquidity Position is stored in the Asset Module,
        // not fetched from the NonfungiblePositionManager.
        // Since liquidity of a position can be increased by a non-owner,
        // the max exposure checks could otherwise be circumvented.
>       assetToLiquidity[assetId] = liquidity;

        bytes32 assetKey = _getKeyFromAsset(address(NON_FUNGIBLE_POSITION_MANAGER),
        assetId);
        bytes32[] memory underlyingAssetKeys = new bytes32[](2);
        underlyingAssetKeys[0] = _getKeyFromAsset(token0, 0);
        underlyingAssetKeys[1] = _getKeyFromAsset(token1, 0);
        assetToUnderlyingAssets[assetKey] = underlyingAssetKeys;
    }
```

Similarly, when withdrawing, the record is deleted from the mapping.

```
## UniswapV3AssetModule.sol

    function processDirectWithdrawal(address creditor, address asset, uint256
    assetId, uint256 amount)
        public
        override
        returns (uint256 assetType)
    {
        assetType = super.processDirectWithdrawal(creditor, asset, assetId, amount);

        // If the asset is withdrawn, remove its from the mapping.
        // If we keep the liquidity of the asset in storage,
        // an offchain getValue of the asset will be calculated with the stored
        liquidity.
>       delete assetToLiquidity[assetId];
    }
```

However, the current implementation doesn't account for the cases where you can withdraw `amount = 0`. And as shown above, that would still delete the liquidity entry.

A user can deposit a Uniswap V3 position with a small amount of liquidity that passes the exposure check, withdraw the zero amount and then add new liquidity to the position that gets taken into account when the position value is calculated.

Coded, runnable PoC:

```
function testFuzz_Success_processDirectWithdrawal(
        uint128 liquidity,
        int24 tickLower,
        int24 tickUpper,
        uint112 maxUsdExposureProtocol,
        uint256 priceToken0,
        uint256 priceToken1,
        uint112 initialExposure0,
        uint112 initialExposure1,
        uint112 maxExposure0,
        uint112 maxExposure1
    ) public {
        vm.assume(tickLower < tickUpper);
        vm.assume(isWithinAllowedRange(tickLower));
        vm.assume(isWithinAllowedRange(tickUpper));

        vm.assume(liquidity > 0);

        // Calculate and check that tick current is within allowed ranges.
        uint160 sqrtPriceX96 =
        uint160(calculateAndValidateRangeTickCurrent(priceToken0, priceToken1));
        vm.assume(isWithinAllowedRange(TickMath.getTickAtSqrtRatio(sqrtPriceX96)));

        // Create Uniswap V3 pool initiated at tickCurrent with cardinality 300.
        pool = createPool(token0, token1,
        TickMath.getSqrtRatioAtTick(TickMath.getTickAtSqrtRatio(sqrtPriceX96)), 300);

        // Check that Liquidity is within allowed ranges.
        vm.assume(liquidity <= pool.maxLiquidityPerTick());

        // Mint liquidity position.
        uint256 tokenId = addLiquidity(pool, liquidity, users.liquidityProvider,
        tickLower, tickUpper, false);

        // Hacky way to avoid stack to deep.
        int24[] memory ticks = new int24[](3);
        ticks[0] = TickMath.getTickAtSqrtRatio(sqrtPriceX96);
        ticks[1] = tickLower;
        ticks[2] = tickUpper;

        uint128 liquidity_;
        {
            // Calculate amounts of underlying tokens.
            // We do not use the fuzzed liquidity, but fetch liquidity from the
            contract.
            // This is because there might be some small differences due to rounding
            errors.
            (,,,,,,, liquidity_,,,,) = nonfungiblePositionManager.positions(tokenId);
```

```
            (uint256 amount0, uint256 amount1) =
            LiquidityAmounts.getAmountsForLiquidity(
                sqrtPriceX96, TickMath.getSqrtRatioAtTick(ticks[1]),
                TickMath.getSqrtRatioAtTick(ticks[2]), liquidity_
            );

            // Check that exposure to underlying tokens stays below maxExposures.
            vm.assume(amount0 + initialExposure0 < maxExposure0);
            vm.assume(amount1 + initialExposure1 < maxExposure1);

            // And: Usd value of underlying assets does not overflow.
            vm.assume(amount0 + initialExposure0 <= type(uint256).max / priceToken0 /
            10 ** (18 - 0)); // divided by 10 ** (18 - DecimalsOracle).
            vm.assume(amount1 + initialExposure1 <= type(uint256).max / priceToken0 /
            10 ** (18 - 0)); // divided by 10 ** (18 - DecimalsOracle).
        }

        // Add underlying tokens and its oracles to Arcadia.
        addUnderlyingTokenToArcadia(address(token0), int256(uint256(priceToken0)),
        initialExposure0, maxExposure0);
        addUnderlyingTokenToArcadia(address(token1), int256(uint256(priceToken1)),
        initialExposure1, maxExposure1);

        {
            // And: usd exposure to protocol below max usd exposure.
            (uint256 usdExposureProtocol,,) =
                uniV3AssetModule.getValue(address(creditorUsd),
                address(nonfungiblePositionManager), tokenId, 1);
            vm.assume(usdExposureProtocol < type(uint112).max);
            maxUsdExposureProtocol = uint112(bound(maxUsdExposureProtocol,
            usdExposureProtocol + 1, type(uint112).max));
        }

        vm.prank(users.riskManager);
        registryExtension.setRiskParametersOfDerivedAssetModule(
            address(creditorUsd), address(uniV3AssetModule), maxUsdExposureProtocol,
            100
        );

        vm.prank(address(registryExtension));
        uniV3AssetModule.processDirectDeposit(address(creditorUsd),
        address(nonfungiblePositionManager), tokenId, 1);

        vm.prank(address(registryExtension));
        uniV3AssetModule.processDirectWithdrawal(address(creditorUsd),
        address(nonfungiblePositionManager), tokenId, 0);

        assertEq(uniV3AssetModule.getAssetToLiquidity(tokenId), 0);
    }
```

Recommendation:

```
 ## UniswapV3AssetModule.sol

@@ -455,12 +455,17 @@ contract UniswapV3AssetModule is DerivedAssetModule {
         override
         returns (uint256 assetType)
     {
+        bytes32 assetKey = _getKeyFromAsset(asset, assetId);
+        uint112 exposureAssetBefore =
lastExposuresAsset[creditor][assetKey].lastExposureAsset;
+
         assetType = super.processDirectWithdrawal(creditor, asset, assetId, amount);

         // If the asset is withdrawn, remove its from the mapping.
         // If we keep the liquidity of the asset in storage,
         // an offchain getValue of the asset will be calculated with the stored
         liquidity.
-        delete assetToLiquidity[assetId];
+        if (lastExposuresAsset[creditor][assetKey].lastExposureAsset == 0 &&
exposureAssetBefore > 0) {
+            delete assetToLiquidity[assetId];
+        }
     }

     /**
@@ -482,6 +487,9 @@ contract UniswapV3AssetModule is DerivedAssetModule {
         uint256 exposureUpperAssetToAsset,
         int256 deltaExposureUpperAssetToAsset
     ) public override returns (uint256 usdExposureUpperAssetToAsset) {
+        bytes32 assetKey = _getKeyFromAsset(asset, assetId);
+        uint112 exposureAssetBefore =
lastExposuresAsset[creditor][assetKey].lastExposureAsset;
+
         usdExposureUpperAssetToAsset = super.processIndirectWithdrawal(
             creditor, asset, assetId, exposureUpperAssetToAsset,
             deltaExposureUpperAssetToAsset
         );
@@ -489,6 +497,8 @@ contract UniswapV3AssetModule is DerivedAssetModule {
         // If the asset is withdrawn, remove its from the mapping.
         // If we keep the liquidity of the asset in storage,
         // an offchain getValue of the asset will be calculated with the stored
         liquidity.
-        delete assetToLiquidity[assetId];
+        if (lastExposuresAsset[creditor][assetKey].lastExposureAsset == 0 &&
exposureAssetBefore > 0) {
+            delete assetToLiquidity[assetId];
+        }
     }
 }
```

In our discussions with the client, it has been determined that updating the `liquidity` on zero amount deposits is also unintended and a fix for both issues has been applied during the audit.

Arcadia: Addressed in PR 141.

Renascence: The issue has been fixed.

Still, there are two things worth noting.

One, there is no check in `UniswapV3AssetModule.processIndirectWithdrawal()` that if `deltaExpo-`

`sureUpperAssetToAsset != -1` it must be equal to zero. This is inconsistent with `UniswapV3AssetModule.processIndirectDeposit()`. The purpose of it is that in the case of a corrupted asset module that passes `deltaExposureUpperAssetToAsset = 1`, withdrawals are still allowed and the account can get liquidated if needed.

Second, it is in theory possible that another asset module would have the `UniswapV3AssetModule` as one of its underlying assets. Since `UniswapV3AssetModule.processIndirectWithdrawal()` doesn't work correctly for a positive exposure delta and `UniswapV3AssetModule.processIndirectDeposit()` doesn't work correctly for a negative exposure delta, the `UniswapV3AssetModule` wouldn't be compatible with a rebasing upstream asset module.

[M-4] Calling `batchProcessDeposit()` before `batchProcessWithdrawal()` allows bypassing exposure limits

Context: AccountV1.sol

Impact: Exposure limits for Uniswap V3 positions can be bypassed. The exposure limits are a defense mechanism against price manipulation attacks and so bypassing them would allow an attacker to deposit large amounts of inflated assets to then steal other assets.

Description: When a user deposits assets through `UniswapV3AssetModule.processDirectDeposit()`, it first makes a call to `UniswapV3AssetModule._addAsset()`

```
## UniswapV3AssetModule.sol

 function processDirectDeposit(address creditor, address asset, uint256 assetId,
 uint256 amount)
        public
        override
        returns (uint256, uint256)
    {
        // For uniswap V3 every id is a unique asset -> on every deposit the asset
        must added to the Asset Module.
        _addAsset(assetId);

        // Also checks that msg.sender == Registry.
        return super.processDirectDeposit(creditor, asset, assetId, amount);
    }
```

`UniswapV3AssetModule._addAsset()` fetches the liquidity of the position from the `NonfungiblePositionManager` and then records it in `assetToLiquidity`.

```
## UniswapV3AssetModule.sol

    function _addAsset(uint256 assetId) internal {
        if (assetId > type(uint96).max) revert InvalidId();
>       (,, address token0, address token1,,,, uint128 liquidity,,,,) =
NON_FUNGIBLE_POSITION_MANAGER.positions(assetId);
        if (liquidity == 0) revert ZeroLiquidity();
>       assetToLiquidity[assetId] = liquidity;
    }
```

When processing a direct withdrawal, the stored liquidity is deleted in `UniswapV3AssetModule.processDirectWithdrawal()`.

```
    function processDirectWithdrawal(address creditor, address asset, uint256
    assetId, uint256 amount)
        public
        override
        returns (uint256 assetType)
    {
        assetType = super.processDirectWithdrawal(creditor, asset, assetId, amount);

        // If the asset is withdrawn, remove its from the mapping.
        // If we keep the liquidity of the asset in storage,
        // an offchain getValue of the asset will be calculated with the stored
        liquidity.
>       delete assetToLiquidity[assetId];
    }
```

The vulnerability exists when creditors are switched. By performing the deposit with the new creditor before the withdrawal with the old creditor, the value in the `assetToLiquidity` mapping is deleted.

Thereby, when the value of the Uniswap V3 position is queried, the current liquidity is queried instead of using stored liquidity.

This bypasses the risk control of creditors. They will be able to take on more risks than they should be able to.

A user can deposit a Uniswap V3 position with a small amount of liquidity that passes the exposure check, perform the creditor switch, and then add new liquidity to the position that gets taken into account when the position value is calculated.

The issue exists in `AccountV1.openMarginAccount()` and `AccountV1.flashActionByCreditor()`.

```
## AccountV1.sol

 function openMarginAccount(address newCreditor)
        external
        onlyOwner
        nonReentrant
        notDuringAuction
        updateActionTimestamp
    {   ...
>       IRegistry(registry).batchProcessDeposit(newCreditor, assetAddresses,
assetIds, assetAmounts);

        // Remove the exposures of the Account for the old Creditor.
>       if (oldCreditor != address(0)) {
>           IRegistry(registry).batchProcessWithdrawal(oldCreditor, assetAddresses,
assetIds, assetAmounts);
        }
```

```
## AccountV1.sol

function flashActionByCreditor(address actionTarget, bytes calldata actionData)
        external
        nonReentrant
        notDuringAuction
        updateActionTimestamp
        returns (uint256 accountVersion)
    {    ...
>           IRegistry(registry).batchProcessDeposit(msg.sender, assetAddresses,
assetIds, assetAmounts);

            // Remove the exposures of the Account for the current Creditor.
            if (currentCreditor != address(0)) {
>               IRegistry(registry).batchProcessWithdrawal(currentCreditor,
assetAddresses, assetIds, assetAmounts);
            }
        }
```

Recommendation: Perform the withdrawal before the deposit.

```
 ## AccountV1.sol

@@ -322,15 +322,15 @@ contract AccountV1 is AccountStorageV1, IAccount {
         address oldCreditor = creditor;
         if (oldCreditor == newCreditor) revert AccountErrors.CreditorAlreadySet();

-        // Check if all assets in the Account are allowed by the new Creditor
-        // and add the exposure of the account for the new Creditor.
-        IRegistry(registry).batchProcessDeposit(newCreditor, assetAddresses,
assetIds, assetAmounts);
-
         // Remove the exposures of the Account for the old Creditor.
         if (oldCreditor != address(0)) {
             IRegistry(registry).batchProcessWithdrawal(oldCreditor, assetAddresses,
             assetIds, assetAmounts);
         }

+        // Check if all assets in the Account are allowed by the new Creditor
+        // and add the exposure of the account for the new Creditor.
+        IRegistry(registry).batchProcessDeposit(newCreditor, assetAddresses,
assetIds, assetAmounts);
+
         // Open margin account for the new Creditor.
         _openMarginAccount(newCreditor);

@@ -731,15 +731,15 @@ contract AccountV1 is AccountStorageV1, IAccount {
             (address[] memory assetAddresses, uint256[] memory assetIds, uint256[]
             memory assetAmounts) =
                 generateAssetData();

-            // Check if all assets in the Account are allowed by the approved
Creditor
-            // and add the exposure of the account for the approved Creditor.
-            IRegistry(registry).batchProcessDeposit(msg.sender, assetAddresses,
assetIds, assetAmounts);
-
             // Remove the exposures of the Account for the current Creditor.
             if (currentCreditor != address(0)) {
                 IRegistry(registry).batchProcessWithdrawal(currentCreditor,
                 assetAddresses, assetIds, assetAmounts);
             }

+            // Check if all assets in the Account are allowed by the approved
Creditor
+            // and add the exposure of the account for the approved Creditor.
+            IRegistry(registry).batchProcessDeposit(msg.sender, assetAddresses,
assetIds, assetAmounts);
+
             // Open margin account for the approved Creditor.
             _openMarginAccount(msg.sender);
         }
```

Arcadia: Addressed in PR 130.

Renascence: The recommendation has been implemented.

[M-5] UniswapV3AssetModule calculates underlying asset amounts irrespective of amount parameter

Context: UniswapV3AssetModule.sol

Impact: Exposure of creditors in `UniswapV3AssetModule` can be manipulated to the upside. This results in a DoS attack where further deposits are blocked by maxing out the exposure of a `Creditor`.

Description: The `UniswapV3AssetModule._getUnderlyingAssetsAmounts()` function calculates the amounts of underlying tokens for a Uniswap V3 position.

```
## UniswapV3AssetModule.sol

    function _getUnderlyingAssetsAmounts(address creditor, bytes32 assetKey, uint256,
    bytes32[] memory)
        internal
        view
        override
        returns (uint256[] memory underlyingAssetsAmounts, AssetValueAndRiskFactors[]
        memory rateUnderlyingAssetsToUsd)
    {
        (, uint256 assetId) = _getAssetFromKey(assetKey);


        (address token0, address token1, int24 tickLower, int24 tickUpper, uint128
        liquidity) = _getPosition(assetId);


        // Get the trusted rates to USD of the Underlying Assets.
        bytes32[] memory underlyingAssetKeys = new bytes32[](2);
        underlyingAssetKeys[0] = _getKeyFromAsset(token0, 0);
        underlyingAssetKeys[1] = _getKeyFromAsset(token1, 0);
        rateUnderlyingAssetsToUsd = _getRateUnderlyingAssetsToUsd(creditor,
        underlyingAssetKeys);
        ...
```

The function's third parameter is the `amount`, but it is never used. This means that even if `amount=0` is passed, the amounts of underlying assets are calculated as though `amount` was equal to `1`.

This allows for two scenarios:

1. The exposure isn't reduced when a legitimate withdrawal (`amount=0`) occurs
2. An attacker can deposit zero amounts of positions, and the exposure IS updated

Both scenarios result in the same impact: an inflated exposure of the `Creditor` to the Uniswap V3 protocol and the underlying assets.

The following PoC illustrates the exposure manipulation by depositing a zero amount of a Uniswap V3 position.

```
function testFuzz_Success_processDirectDepositUniV3(
        uint128 liquidity,
        int24 tickLower,
        int24 tickUpper,
        uint112 maxUsdExposureProtocol,
```

```
        uint256 priceToken0,
        uint256 priceToken1,
        uint112 initialExposure0,
        uint112 initialExposure1,
        uint112 maxExposure0,
        uint112 maxExposure1
    ) public {
        vm.assume(tickLower < tickUpper);
        vm.assume(isWithinAllowedRange(tickLower));
        vm.assume(isWithinAllowedRange(tickUpper));

        vm.assume(liquidity > 0);

        // Calculate and check that tick current is within allowed ranges.
        uint160 sqrtPriceX96 =
        uint160(calculateAndValidateRangeTickCurrent(priceToken0, priceToken1));
        vm.assume(isWithinAllowedRange(TickMath.getTickAtSqrtRatio(sqrtPriceX96)));

        // Create Uniswap V3 pool initiated at tickCurrent with cardinality 300.
        pool = createPool(token0, token1,
        TickMath.getSqrtRatioAtTick(TickMath.getTickAtSqrtRatio(sqrtPriceX96)), 300);

        // Check that Liquidity is within allowed ranges.
        vm.assume(liquidity <= pool.maxLiquidityPerTick());

        // Mint liquidity position.
        uint256 tokenId = addLiquidity(pool, liquidity, users.liquidityProvider,
        tickLower, tickUpper, false);

        // Hacky way to avoid stack to deep.
        int24[] memory ticks = new int24[](3);
        ticks[0] = TickMath.getTickAtSqrtRatio(sqrtPriceX96);
        ticks[1] = tickLower;
        ticks[2] = tickUpper;

        {
            // Calculate amounts of underlying tokens.
            // We do not use the fuzzed liquidity, but fetch liquidity from the
            contract.
            // This is because there might be some small differences due to rounding
            errors.
            (,,,,,,, uint128 liquidity_,,,,) =
            nonfungiblePositionManager.positions(tokenId);
            (uint256 amount0, uint256 amount1) =
            LiquidityAmounts.getAmountsForLiquidity(
                sqrtPriceX96, TickMath.getSqrtRatioAtTick(ticks[1]),
                TickMath.getSqrtRatioAtTick(ticks[2]), liquidity_
            );

            // Check that exposure to underlying tokens stays below maxExposures.
            vm.assume(amount0 + initialExposure0 < maxExposure0);
            vm.assume(amount1 + initialExposure1 < maxExposure1);

            // And: Usd value of underlying assets does not overflow.
            vm.assume(amount0 + initialExposure0 <= type(uint256).max / priceToken0 /
            10 ** (18 - 0)); // divided by 10 ** (18 - DecimalsOracle).
            vm.assume(amount1 + initialExposure1 <= type(uint256).max / priceToken0 /
            10 ** (18 - 0)); // divided by 10 ** (18 - DecimalsOracle).
        }
```

```
        // Add underlying tokens and its oracles to Arcadia.
        addUnderlyingTokenToArcadia(address(token0), int256(uint256(priceToken0)),
        initialExposure0, maxExposure0);
        addUnderlyingTokenToArcadia(address(token1), int256(uint256(priceToken1)),
        initialExposure1, maxExposure1);

        {
            // And: usd exposure to protocol below max usd exposure.
            (uint256 usdExposureProtocol,,) =
                uniV3AssetModule.getValue(address(creditorUsd),
                address(nonfungiblePositionManager), tokenId, 1);
            vm.assume(usdExposureProtocol < type(uint112).max);
            maxUsdExposureProtocol = uint112(bound(maxUsdExposureProtocol,
            usdExposureProtocol + 1, type(uint112).max));
        }

        vm.prank(users.riskManager);
        registryExtension.setRiskParametersOfDerivedAssetModule(
            address(creditorUsd), address(uniV3AssetModule), maxUsdExposureProtocol,
            100
        );

        (uint112 lastUsdExposureProtocolBefore,,) =
        uniV3AssetModule.riskParams(address(creditorUsd));
        vm.prank(address(registryExtension));
        (uint256 recursiveCalls, uint256 assetType) =
            uniV3AssetModule.processDirectDeposit(address(creditorUsd),
            address(nonfungiblePositionManager), tokenId, 0);

        assertEq(recursiveCalls, 3);
        assertEq(assetType, 1);
        (uint112 lastUsdExposureProtocolAfter,,) =
        uniV3AssetModule.riskParams(address(creditorUsd));
        assertGt(lastUsdExposureProtocolAfter, lastUsdExposureProtocolBefore);
    }
```

Recommendation: Return zero amounts if `amount=0`. Note that the `rateUnderlyingAssetsToUsd` array can remain empty. In this case, the rates will be queried downstream if needed.

```
## UniswapV3AssetModule.sol

function _getUnderlyingAssetsAmounts(address creditor, bytes32 assetKey, uint256
amount, bytes32[] memory)
        internal
        view
        override
        returns (uint256[] memory underlyingAssetsAmounts, AssetValueAndRiskFactors[]
        memory rateUnderlyingAssetsToUsd)
    {
+       if (amount == 0) {
+           return (new uint[](2), rateUnderlyingAssetsToUsd);
+       }
+
        (, uint256 assetId) = _getAssetFromKey(assetKey);

        (address token0, address token1, int24 tickLower, int24 tickUpper, uint128
        liquidity) = _getPosition(assetId);
```

Arcadia: Addressed in PR 133.

Renascence: The recommendation has been implemented.

4.3 Low Risk

[L-1] AccountV1.setAssetManager() should be callable by everyone

Context: AccountV1.sol

Description: Currently the `AccountV1.setAssetManager()` function is protected by the `onlyOwner` modifier.

```
## AccountV1.sol

    function setAssetManager(address assetManager, bool value) external onlyOwner {
        emit AssetManagerSet(msg.sender, assetManager,
        isAssetManager[msg.sender][assetManager] = value);
    }
```

However, this is not ideal, and it should be callable by everybody.

Assume the scenario where you get an Account transferred to you that you have previously owned and set an `assetManager` for.

You got e.g., liquidated, and the `assetManager` wasn't reset.

Getting the Account transferred back would be dangerous since the `assetManager` is still set, and he can perform flash actions.

Removing the access control means you can proactively remove the `assetManager`.

Recommendation:

```
 ## AccountV1.sol

-    function setAssetManager(address assetManager, bool value) external onlyOwner {
+    function setAssetManager(address assetManager, bool value) external {
         emit AssetManagerSet(msg.sender, assetManager,
         isAssetManager[msg.sender][assetManager] = value);
     }
```

Arcadia: Addressed in PR 142.

Renascence: The recommendation has been implemented.

[L-2] LendingPool.liquidityOf() should round down instead of up

Context: LendingPool.sol

Description: The rounding in `LendingPool.liquidityOf()` should be down instead of up.

```
## LendingPool.sol

   function liquidityOf(address owner_) external view returns (uint256 assets) {
        // Avoid a second calculation of unrealised debt (expensive).
        // if interests are already synced this block.
        if (lastSyncedTimestamp != uint32(block.timestamp)) {
            // The total liquidity of a tranche equals the sum of the realised
            liquidity
            // of the tranche, and its pending interests.
>           uint256 interest = calcUnrealisedDebt().mulDivUp(interestWeight[owner_],
totalInterestWeight);
            unchecked {
                assets = realisedLiquidityOf[owner_] + interest;
            }
        } else {
            assets = realisedLiquidityOf[owner_];
        }
    }
```

The rounding is inaccurate since the `LendingPool.liquidityOfAndSync()` function, which is used for the state-changing calculations, rounds down. The wrong rounding impacts the following functions:

```
## Tranche.sol

    function maxMint(address) public view override returns (uint256 maxShares)
    function maxWithdraw(address owner_) public view override returns (uint256
    maxAssets)
    function maxRedeem(address owner_) public view override returns (uint256
    maxShares)
```

Note that they are only used externally and don't affect the internal state, hence the ”Low” severity.

Recommendation:

```
 ## LendingPool.sol

        if (lastSyncedTimestamp != uint32(block.timestamp)) {
            // The total liquidity of a tranche equals the sum of the realised
            liquidity
            // of the tranche, and its pending interests.
-           uint256 interest = calcUnrealisedDebt().mulDivUp(interestWeight[owner_],
totalInterestWeight);
+           uint256 interest =
calcUnrealisedDebt().mulDivDown(interestWeight[owner_], totalInterestWeight);
            unchecked {
                assets = realisedLiquidityOf[owner_] + interest;
            }
```

Arcadia: Addressed in PR 105.

Renascence: The recommendation has been implemented.

[L-3] settleAuction() doesnt account for fixedLiquidationCost

Context: Liquidator.sol

Description: The check should be `if (collateralValue >= usedMargin || ILendingPool(creditor).getOpenPosition(account) == 0)`.

```
## Liquidator.sol

 function _settleAuction(address account, AuctionInformation storage
 auctionInformation_)
        internal
        returns (bool success)
    {
        ...
        uint256 collateralValue = IAccount(account).getCollateralValue();
        ...

        // Check the different conditions to end the auction.
>       if (collateralValue >= usedMargin) {
            // Happy flow: Account is back in a healthy state.
            ILendingPool(creditor).settleLiquidationHappyFlow(account, startDebt,
            msg.sender);
        } else if (collateralValue == 0) {
```

That's because `usedMargin` includes `fixedLiquidationCost`. Hence, if someone repays the whole loan directly via the `Creditor`, the function can just execute the happy flow even though `collateralValue` might be smaller than the `fixedLiquidatonCost`.

In the worst case, the auction is settled with the third case which transfers account ownership to the account recipient.

```
## Liquidator.sol

        } else if (block.timestamp > auctionInformation_.cutoffTimeStamp) {
            // Unhappy flow: Auction did not end within the cutoffTime.
            ILendingPool(creditor).settleLiquidationUnhappyFlow(account, startDebt,
            msg.sender);
            // All remaining assets are transferred to the asset recipient,
            // and a manual (trusted) liquidation has to be done.
            IAccount(account).auctionBoughtIn(creditorToAccountRecipient[creditor]);
        } else {
```

Note that there's no loss of funds since there's no bad debt and the account recipient is trusted and so he will transfer the Account back. Last but not least the `collateralValue` of the Account must have been below the `fixedLiquidationCost`. Therefore, the ”Low” severity.

Recommendation:

```
 ## Liquidator.sol

         // Check the different conditions to end the auction.
-        if (collateralValue >= usedMargin) {
+        if (collateralValue >= usedMargin ||
ILendingPool(creditor).getOpenPosition(account) == 0) {
             // Happy flow: Account is back in a healthy state.
             ILendingPool(creditor).settleLiquidationHappyFlow(account, startDebt,
             msg.sender);
         } else if (collateralValue == 0) {
```

Arcadia: Addressed in PR 115.

Renascence: The implemented `usedMargin == minimumMargin` check is equivalent to the proposed `getOpenPosition() == 0` check.

[L-4] Tranche interestRate can be used for inflation attack

Context: LendingPool.sol, Tranche.sol

Description: The comments say that `Tranche` isn't vulnerable to inflation attacks.

```
## Tranche.sol
 /*
> * @dev Implementation not vulnerable to ERC4626 inflation attacks,
> * since totalAssets() cannot be manipulated by first minter when total amount of
shares are low.
  * For more information, see
  https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3706
  */
 contract Tranche is ITranche, ERC4626, Owned {
```

However, assets in the `Tranche` Vault are represented in the `realisedLiquidityOf[tranche]` mapping of the `LendingPool`.

Notice the comment in `LendingPool.donateToTranche()` that since you cannot donate to a `Tranche` with dust amount of shares, there is no inflation attack possible.

```
## LendingPool.sol
     /*
      * @dev Can be used by anyone to donate assets to the Lending Pool.
      * It is supposed to serve as a way to compensate the jrTranche after an
      * auction didn't get sold and was manually liquidated after cutoffTime.
>     * @dev First minter of a tranche could abuse this function by minting only 1
share,
>     * frontrun next minter by calling this function and inflate the share price.
>     * This is mitigated by checking that there are at least 10 ** decimals shares
outstanding.
      */
     function donateToTranche(uint256 trancheIndex, uint256 assets) external
     whenDepositNotPaused processInterests {
         ...
         if (ERC4626(tranche).totalSupply() < 10 ** decimals) revert
         LendingPoolErrors.InsufficientShares();
```

However, the interest payments for the `Tranche` can be used with the same effect.

```
## LendingPool.sol

    function _syncInterestsToLiquidityProviders(uint256 assets) internal {
        ...
        for (uint256 i; i < trancheLength; ++i) {
            trancheShare = assets.mulDivDown(interestWeightTranches[i],
            totalInterestWeight_);
            unchecked {
>               realisedLiquidityOf[tranches[i]] += trancheShare;
```

For example, if a first depositor deposits `1 wei` and gets minted `1 share`. Then, the interest is is e.g., `1e18 wei`, meaning there is roughly a `1 share` to `1e18 wei` conversion rate.

Then, assume a second depositor deposits `2e18 wei`; he only gets `1 share` and loses by approximation `0.5e18 wei` to the first deposit.

The impact of this issue is "Medium", because certain users that deposit to the Tranche at such a time can lose some of their funds.

Still, this is a "Low" severity issue because of it's "Low" likelihood.

A malicious party cannot control this scenario, and it probably wouldn't ever happen when adding a `Tranche` to a fresh `LendingPool`.

Recommendation: Applying the `processInterest` modifier to both functions below will ensure the `Tranche` starts with a zero-interest payment.

```
 ## LendingPool.sol

-    function addTranche(address tranche, uint16 interestWeight_, uint16
liquidationWeight) external onlyOwner {
+    function addTranche(address tranche, uint16 interestWeight_, uint16
liquidationWeight) external onlyOwner processInterests {
         if (auctionsInProgress > 0) revert LendingPoolErrors.AuctionOngoing();
         if (isTranche[tranche]) revert LendingPoolErrors.TrancheAlreadyExists();


-    function setTrancheWeights(uint256 index, uint16 interestWeight_, uint16
liquidationWeight) external onlyOwner {
+    function setTrancheWeights(uint256 index, uint16 interestWeight_, uint16
liquidationWeight) external onlyOwner processInterests {
         if (index >= tranches.length) revert LendingPoolErrors.NonExistingTranche();
         totalInterestWeight = totalInterestWeight - interestWeightTranches[index] +
         interestWeight_;
         interestWeightTranches[index] = interestWeight_;
```

An additional measure of safety is to have new `Tranche`s start with a low or zero interest weight and to only increase the interest weight when there is sufficient activity and shares supply in the `Tranche`.

Arcadia: Addressed in PR 106.

Renascence: The recommendation has been implemented.

[L-5] UniswapV3AssetModule.isAllowed() should check if liquidity is greater than zero

Context: UniswapV3AssetModule.sol

Description: Since `UniswapV3AssetModule.isAllowed()` does not include a check that `liquidity > 0`, it is possible that a user deposits a Uniswap V3 position with zero liquidity into his `Account` if he doesn't have a `creditor` set.

```
## UniswapV3AssetModule.sol

  function isAllowed(address asset, uint256 assetId) public view override returns
  (bool) {
        if (asset != address(NON_FUNGIBLE_POSITION_MANAGER)) return false;

        try NON_FUNGIBLE_POSITION_MANAGER.positions(assetId) returns (
            uint96,
            address,
            address token0,
            address token1,
            uint24,
            int24,
            int24,
            uint128,
            uint256,
            uint256,
            uint128,
            uint128
        ) {
            return IRegistry(REGISTRY).isAllowed(token0, 0) &&
            IRegistry(REGISTRY).isAllowed(token1, 0);
        } catch {
            return false;
        }
    }
```

However, when the user wants to set a `creditor`, the `Registry.batchProcessDeposit()` call will **fail** as it relies on `UniswapV3AssetModule_addAsset()`, which does have the check that liquidity must be greater than zero:

```
## UniswapV3AssetModule.sol

  function _addAsset(uint256 assetId) internal {
        if (assetId > type(uint96).max) revert InvalidId();

        (,, address token0, address token1,,,, uint128 liquidity,,,,) =
        NON_FUNGIBLE_POSITION_MANAGER.positions(assetId);

        // No need to explicitly check if token0 and token1 are allowed, _addAsset()
        is only called in the
        // deposit functions and there any deposit of non-allowed Underlying Assets
        will revert.
>       if (liquidity == 0) revert ZeroLiquidity();
          ...
    }
```

Recommendation:

```
 ## UniswapV3AssetModule.sol

  function isAllowed(address asset, uint256 assetId) public view override returns
  (bool) {
        if (asset != address(NON_FUNGIBLE_POSITION_MANAGER)) return false;

        try NON_FUNGIBLE_POSITION_MANAGER.positions(assetId) returns (
            uint96,
            address,
            address token0,
            address token1,
            uint24,
            int24,
            int24,
-            uint128,
+            uint128 liquidity,
            uint256,
            uint256,
            uint128,
            uint128
        ) {
-            return IRegistry(REGISTRY).isAllowed(token0, 0) &&
IRegistry(REGISTRY).isAllowed(token1, 0);
+            return IRegistry(REGISTRY).isAllowed(token0, 0) &&
IRegistry(REGISTRY).isAllowed(token1, 0) && liqudity > 0;
        } catch {
            return false;
        }
    }
```

Arcadia: Addressed in PR 144.

Renascence: The recommendation has been implemented.

[L-6] Accounts cannot be liquidated if no Tranches are set

Context: LendingPool.sol

Description: When calling `LendingPool.startLiquidation()`, the line below reverts if there are no Tranches.

```
## LendingPool.sol

    function startLiquidation(address initiator)
        external
        override
        whenLiquidationNotPaused
        processInterests
        returns (uint256 startDebt)
    {
        ...
>       if (auctionsInProgress == 0) ITranche(tranches[tranches.length -
1]).setAuctionInProgress(true);
```

Based on that, we can conclude that there's the assumption that when there are no Tranches, there

is also no debt to be paid back.

However, that's wrong because besides Tranches, multiple other parties can have deposits in the `LendingPool`:

- Terminators
- Initiators
- Account owners
- Treasury

This leads us to the edge case, where if there are remaining deposits and loans, the loans won't be able to get liquidated because there are no Tranches, leading to a temporary freeze of funds.

However, the finding's severity is "Low" because the `LendingPool` owner can easily add a new `Tranche` with zero deposits to resolve the situation.

Recommendation:

```
 ## LendingPool.sol

@@ -882,7 +882,7 @@ contract LendingPool is LendingPoolGuardian, Creditor, DebtToken,
ILendingPool {
         totalRealisedLiquidity = SafeCastLib.safeCastTo128(totalRealisedLiquidity +
         initiationReward);

         // If this is the sole ongoing auction, prevent any deposits and withdrawals
         in the most jr tranche
-        if (auctionsInProgress == 0) ITranche(tranches[tranches.length -
1]).setAuctionInProgress(true);
+        if (auctionsInProgress == 0 && tranches.length > 0)
ITranche(tranches[tranches.length - 1]).setAuctionInProgress(true);

         unchecked {
             ++auctionsInProgress;
```

Arcadia: Addressed in PR 107.

Renascence: The recommendation has been implemented.

[L-7] Incorrect interest rate while auctions are ongoing

Context: LendingPool.sol

Description: The `LendingPool.skim()` function cannot be called when auctions are ongoing. This makes sense since `realisedDebt` and `totalRealisedLiquidity` are "out-of-sync" during liquidations.

But the same problem occurs for the interest rate for which `realisedDebt / totalRealisedLiquidity` is the utilization ratio.

In `LendingPool.startLiquidation()`, the `realisedDebt` is increased by the full liquidation cost

- `(_deposit(initiationReward + liquidationPenalty + terminationReward, msg.sender))`

while `totalRealisedLiquidity` is only increased by `initiationReward`.

Thereby, the utilization and the interest rate are increased unfairly.

It's unfair because

1. `totalRealisedLiquidity` isn't increased by `liquidationPenalty` + `terminationReward`
2. `liquidationPenalty` and `terminationReward` isn't even actual debt. They are not "at risk of not being paid." They won't be paid out if the liquidation cannot pay for them. Hence, it doesn't make sense to raise the interest rate.

The severity is "Low" because the issue results in only a small amount of excess interest. Here are two scenarios:

The worst-case scenario is:

- the account that is liquidated utilizes 100% of the pool's funds
- total liquidation penalty equals the maximum of 11%
- high slope is type(uint72).max (~4.722e21)
- cutoffTime is maximum: 18 hours

→ utilisation increases 11%

→ (yearly) interest rate increases by 52.000% (4.722e21 * 0.11 = ~520e18)

→ Which is an interest payment of `1.29%` after 18 hours ((1+520)^(18 hours / 8760 hours per year))

A more realistic scenario is:

- the account that is liquidated utilizes 10% of the pool's funds
- total liquidation penalty equals 5%
- high slope is 1e18 (100% interest per year)
- cutoffTime is maximum: 18 hours

→ utilisation increases 5% * 10% = 0.5%

→ (yearly) interest rate increases by 5% (1e18 * 0.005 = 0.005e18)

→ Which is an interest payment of `0.369%` after 18 hours ((1+5)^(18 hours / 8760 hours per year))

The numbers in the more realistic scenario are still quite high assumptions, hence the "Low" severity.

Recommendation:

```
 ## LendingPool.sol

@@ -126,6 +126,8 @@ contract LendingPool is LendingPoolGuardian, Creditor, DebtToken,
ILendingPool {
     // Stores the credit allowances for a beneficiary per Account and per Owner.
     mapping(address => mapping(address => mapping(address => uint256))) public
     creditAllowance;

+    uint256 internal liquidationDebt;
+
     /* ////////////////////////////////////////////////////////////////
                                 EVENTS
     //////////////////////////////////////////////////////////////// */
@@ -194,7 +196,7 @@ contract LendingPool is LendingPoolGuardian, Creditor, DebtToken,
ILendingPool {
```

```
         _;
         //_updateInterestRate() modifies the state (effect), but can safely be called
         after interactions.
         //Cannot be exploited by re-entrancy attack.
-        _updateInterestRate(realisedDebt, totalRealisedLiquidity);
+        _updateInterestRate(realisedDebt - liquidationDebt, totalRealisedLiquidity);
     }

     /* ////////////////////////////////////////////////////////////// 
@@ -731,7 +733,7 @@ contract LendingPool is LendingPoolGuardian, Creditor, DebtToken,
ILendingPool {
             // over a period of 5 years.
             // This won't overflow as long as openDebt <
             340282366920938491299511414659481 6
             // which is 3.4 million billion *10**18 decimals.
-            unrealisedDebt = (realisedDebt * (LogExpMath.pow(base, exponent) - 1e18))
/ 1e18;
+            unrealisedDebt = ((realisedDebt - liquidationDebt) *
(LogExpMath.pow(base, exponent) - 1e18)) / 1e18;
         }

         return SafeCastLib.safeCastTo128(unrealisedDebt);
@@ -874,6 +876,7 @@ contract LendingPool is LendingPoolGuardian, Creditor, DebtToken,
ILendingPool {

         // Mint the liquidation incentives as extra debt towards the Account.
         _deposit(initiationReward + liquidationPenalty + terminationReward,
         msg.sender);
+        liquidationDebt = liquidationDebt + terminationReward + liquidationPenalty;

         // Increase the realised liquidity for the initiator.
         // The other incentives will only be added as realised liquidity for the
         respective actors
@@ -935,6 +938,7 @@ contract LendingPool is LendingPoolGuardian, Creditor, DebtToken,
ILendingPool {

         // Unsafe cast: sum will revert if it overflows.
         totalRealisedLiquidity = uint128(totalRealisedLiquidity + terminationReward +
         liquidationPenalty + surplus);
+        liquidationDebt = liquidationDebt - terminationReward - liquidationPenalty;

         unchecked {
             // Pay out any surplus to the current Account Owner.
@@ -1005,6 +1009,7 @@ contract LendingPool is LendingPoolGuardian, Creditor,
DebtToken, ILendingPool {
         // Remove the remaining debt from the Account now that it is written off from
         the liquidation incentives/Liquidity Providers.
         _burn(account, shares);
         realisedDebt -= openDebt;
+        liquidationDebt = liquidationDebt - terminationReward - liquidationPenalty;
         emit Withdraw(msg.sender, account, account, openDebt, shares);

         _endLiquidation();
```

Arcadia: The finding has been acknowledged.

[L-8] Liquidations of dust debt positions are unprofitable to initiate which can lead to DoS attempts

Context: LendingPool.sol

Impact: Dust debt positions can be used to DoS liquidations.

Description: It has been determined in communication with the client that Arcadia will run liquidation bots that liquidate unhealthy accounts.

Attackers can create debt positions that cannot be profitably liquidated and even require the liquidation bots to incur costs.

The `initiationReward` (as well as `terminationReward` and `liquidationPenalty`) are calculated as a percentage of an account's `debt` at the start of the liquidation.

```
## LendingPool.sol

    function _calculateRewards(uint256 debt)
        internal
        view
        returns (uint256 initiationReward, uint256 terminationReward, uint256
        liquidationPenalty)
    {
        uint256 maxInitiationFee_ = maxInitiationFee;
        uint256 maxTerminationFee_ = maxTerminationFee;
>       initiationReward = debt.mulDivDown(initiationWeight, ONE_4);
        initiationReward = initiationReward > maxInitiationFee_ ? maxInitiationFee_ :
        initiationReward;


>       terminationReward = debt.mulDivDown(terminationWeight, ONE_4);
        terminationReward = terminationReward > maxTerminationFee_ ?
        maxTerminationFee_ : terminationReward;


>       liquidationPenalty = debt.mulDivUp(penaltyWeight, ONE_4);
    }
```

There is no minimum amount of debt that an account must have and, so it is easy to see that if an account only has a dust amount of debt and becomes unhealthy, a liquidation cannot be profitably initiated.

An attacker could open many dust positions and wait for them to get unhealthy in an attempt to DoS, the liquidators. This would hardly work since it would be associated with a loss when the liquidation is initiated and settled.

That's because the account must hold an additional margin of `fixedLiquidationCost`, ensuring the liquidation is profitable.

```
## AccountV1.sol

    function getUsedMargin() public view returns (uint256 usedMargin) {
        // Cache creditor
        address creditor_ = creditor;
        if (creditor_ == address(0)) return 0;


        // getOpenPosition() is a view function, cannot modify state.
>       usedMargin = ICreditor(creditor_).getOpenPosition(address(this)) +
fixedLiquidationCost;
    }
```

Recommendation: To make the initiation of liquidations profitable, some of the `fixedLiquidationCost` should be used to pay the liquidation initiator.

There are multiple ways to do this. `initiationReward` could be increased by `initiationWeight` percentage of `fixedLiquidationCost`.

What's important is that the `initationReward` makes initiating liquidations profitable, and `fixedLiquidationCost` is set such as to account for the additional funds that go to the initiator.

```
 ## LendingPool.sol

@@ -1116,7 +1116,7 @@ contract LendingPool is LendingPoolGuardian, Creditor,
DebtToken, ILendingPool {
     {
         uint256 maxInitiationFee_ = maxInitiationFee;
         uint256 maxTerminationFee_ = maxTerminationFee;
-        initiationReward = debt.mulDivDown(initiationWeight, ONE_4);
+        initiationReward = (debt + fixedLiquidationCost).mulDivDown(initiationWeight,
ONE_4);
         initiationReward = initiationReward > maxInitiationFee_ ? maxInitiationFee_ :
         initiationReward;

         terminationReward = debt.mulDivDown(terminationWeight, ONE_4);
```

Arcadia: Addressed in PR 115, PR 116, PR 149 and PR 150.

Renascence: Fixing this issue has led to issues with the contract size of the `LendingPool`. As a result, additional optimizations have been made to the `LendingPool`. These have been checked and are safe.

The issue with the unprofitable liquidations has been addressed by introducing a minimum for the `initiationReward` and `terminationReward` which is set as a percentage `x` of `minimumMargin` (previously called `fixedLiquidationCost`).

The `minimumMargin` should be chosen such that `x * minimumMargin` is sufficient to cover gas costs for initiating and terminating liquidations.

If `x` is chosen below `50%` then `100 - 2x` will be available as an additional incentive for bidders. This additional incentive is not necessary but it can be beneficial to allow setting the other risk / liquidation parameters more efficiently.

[L-9] Asset shares in auctions should be rounded up to the bidders disadvantage

Context: Liquidator.sol

Impact: Rounding down in the asset shares calculation in `Liquidator._getAssetShares()` and `Liquidator._calculateTotalShare()` can lead to undervaluing assets. In rare cases assets could be purchased from an auction for free.

Description: Liquidation of an account starts from `Liquidator.liquidateAccount()` and the internal function `Liquidator._getAssetShares()` is responsible for calculating shares for each of the account's assets relative to how much they contribute towards the total value of the account. However, the calculation that determines the asset share rounds down which means we could have assets that contribute a certain amount towards the total value but are assigned a lower amount of shares, therefore, undervaluing the account. In rare cases `assetValues[i].assetValue` can be less than `1/10,000` of `totalValue` which leads to a `0 share` assignment.

```
## Liquidator.sol

    function _getAssetShares(AssetValueAndRiskFactors[] memory assetValues)
        internal
        pure
        returns (uint32[] memory assetShares)
    {
        uint256 length = assetValues.length;
        uint256 totalValue;
        for (uint256 i; i < length; ++i) {
            unchecked {
                totalValue += assetValues[i].assetValue;
            }
        }
        assetShares = new uint32[](length);
        for (uint256 i; i < length; ++i) {
            unchecked {
                // The asset shares are calculated relative to the total value of the
                Account.
                // "assetValue" is a uint256 in Numeraire units, will never overflow.
>               assetShares[i] = uint32(assetValues[i].assetValue * ONE_4 /
totalValue);
            }
        }
    }
```

Similarly, when a user purchases some amount of assets from the auction through `Liquidator.bid()`, a share is calculated depending on the asked asset amounts. This calculation also rounds down and could accumulate a lower share towards `totalShare` and give assets for free to the bidder.

```
## Liquidator.sol

    function _calculateTotalShare(AuctionInformation storage auctionInformation_,
    uint256[] memory askedAssetAmounts)
        internal
        view
        returns (uint256 totalShare)
    {
        uint256[] memory assetAmounts = auctionInformation_.assetAmounts;
        uint32[] memory assetShares = auctionInformation_.assetShares;
        if (assetAmounts.length != askedAssetAmounts.length) {
            revert LiquidatorErrors.InvalidBid();
        }

        // If the AskedAssetAmount is bigger than type(uint224).max, totalShare will
        overflow.
        // However askedAssetAmount can't exceed uint112 in the Account since the
        exposure limits are set to uint112.
        // This means that when the calculated bid price is faulty, the withdraw in
        the Account will always revert.
        for (uint256 i; i < askedAssetAmounts.length; ++i) {
            unchecked {
                totalShare += askedAssetAmounts[i] * assetShares[i] / assetAmounts[i];
            }
        }
    }
```

The fact that a user can split his collateral deposits into many small deposits leads to higher cumulative rounding error than for one single collateral.

Recommendation Round up to disadvantage the bidding user.

```
 ## Liquidator.sol

@@ -255,7 +255,7 @@ contract Liquidator is Owned, ILiquidator {
             unchecked {
                 // The asset shares are calculated relative to the total value of the
                 Account.
                 // "assetValue" is a uint256 in Numeraire units, will never overflow.
-                assetShares[i] = uint32(assetValues[i].assetValue * ONE_4 /
totalValue);
+                assetShares[i] = uint32((assetValues[i].assetValue * ONE_4 +
totalValue - 1) / totalValue);
             }
         }
     }
@@ -331,7 +331,7 @@ contract Liquidator is Owned, ILiquidator {
         // This means that when the calculated bid price is faulty, the withdraw in
         the Account will always revert.
         for (uint256 i; i < askedAssetAmounts.length; ++i) {
             unchecked {
-                totalShare += askedAssetAmounts[i] * assetShares[i] /
assetAmounts[i];
+                totalShare += (askedAssetAmounts[i] * assetShares[i] +
assetAmounts[i] - 1) / assetAmounts[i];
             }
         }
     }
```

Arcadia: Addressed in PR 110.

Renascence: The recommendation has been implemented.

4.4 Informational

[I-1] Withdraw event is emitted with wrong parameter

Context: DebtToken.sol, LendingPool.sol

Description: The `Withdraw` event is emitted with a wrong parameter. The `receiver` of the `asset` is `address(this)`, not `account`.

```
## DebtToken.sol

>    function _withdraw(uint256 assets, address receiver, address account) internal
returns (uint256 shares) {
        // Check for rounding error since we round down in previewWithdraw.
        if ((shares = previewWithdraw(assets)) == 0) revert
        DebtTokenErrors.ZeroShares();

        _burn(account, shares);

        realisedDebt -= assets;

>       emit Withdraw(msg.sender, receiver, account, assets, shares);
    }
```

Recommendation:

```
 ## LendingPool.sol

@@ -498,7 +498,7 @@ contract LendingPool is LendingPoolGuardian, Creditor, DebtToken,
ILendingPool {
         // Address(this) is trusted -> no risk on re-entrancy attack after transfer.
         asset.safeTransferFrom(msg.sender, address(this), amount);

-        _withdraw(amount, account, account);
+        _withdraw(amount, address(this), account);

         emit Repay(account, msg.sender, amount);
     }
@@ -533,7 +533,7 @@ contract LendingPool is LendingPoolGuardian, Creditor, DebtToken,
ILendingPool {
             amount = accountDebt;
         }

-        _withdraw(amount, account, account);
+        _withdraw(amount, address(this), account);

         emit Repay(account, bidder, amount);
     }
```

Arcadia: Addressed in PR 108.

Renascence: The recommendation has been implemented.

[I-2] Remove unused events

Context: Registry.sol, AbstractPrimaryAssetModule.sol

Description: The events are never used and can be removed.

- AllowedActionSet
- MaxExposureSet

Recommendation:

```
 ## Registry.sol

@@ -65,7 +65,6 @@ contract Registry is IRegistry, RegistryGuardian {
                                EVENTS
     ////////////////////////////////////////////////////////////// */

-    event AllowedActionSet(address indexed action, bool allowed);
     event AssetAdded(address indexed assetAddress, address indexed assetModule);
     event AssetModuleAdded(address assetModule);
     event OracleAdded(uint256 indexed oracleId, address indexed oracleModule);

 ## AbstractPrimaryAssetModule.sol

@@ -54,12 +54,6 @@ abstract contract PrimaryAssetModule is AssetModule {
         bytes32 oracleSequence;
     }

-    /* ////////////////////////////////////////////////////////////// */
-                                EVENTS
-    ////////////////////////////////////////////////////////////// */
-
-    event MaxExposureSet(address indexed asset, uint112 maxExposure);
-
     /* ////////////////////////////////////////////////////////////// */
                                ERRORS
     ////////////////////////////////////////////////////////////// */

 ## Events.sol

@@ -64,7 +64,6 @@ abstract contract Events {
                                   REGISTRY
     ////////////////////////////////////////////////////////////////////////////*/

-    event AllowedActionSet(address indexed action, bool allowed);
     event AssetAdded(address indexed assetAddress, address indexed assetModule);
     event AssetModuleAdded(address assetModule);
     event OracleAdded(uint256 indexed oracleId, address indexed oracleModule);
@@ -78,7 +77,6 @@ abstract contract Events {
         uint16 liquidationFactor
     );
-    event MaxExposureSet(address indexed asset, uint128 maxExposure);
```

Arcadia: Addressed in PR 145 and PR-151.

Renascence: The recommendation has been implemented.

[I-3] Unsafe casts of totalRealisedLiquidity

Context: LendingPool.sol

Description

The protocol makes the assumption that `totalRealisedLiquidity` will never reach `type(uint128).max`, reaching it is generally unsafe.

The comments below mention that the two calculations revert on overflow but this is wrong. During addition of `totalRealisedLiquidity` with an `uint256` the result will be stored as an `uint256` which will silently overflow when cast to `uint128`.

```
## LendingPool.sol

   function settleLiquidationUnhappyFlow(address account, uint256 startDebt, address
   terminator)
        external
        whenLiquidationNotPaused
        onlyLiquidator
        processInterests
    {       ...
            // Unsafe cast: sum will revert if it overflows.
            totalRealisedLiquidity = uint128(totalRealisedLiquidity + remainder);
        }
```

```
## LendingPool.sol

    function _settleLiquidationHappyFlow(address account, uint256 startDebt, address
    terminator, uint256 surplus)
        internal
    {   ...
        // Unsafe cast: sum will revert if it overflows.
        totalRealisedLiquidity = uint128(totalRealisedLiquidity + terminationReward +
        liquidationPenalty + surplus);
```

This finding is "Informational" since reaching `type(uint128).max` is unsafe anyways, the different overflow behavior is just a lack of consistency.

Recommendation: For consistency, use `SafeCastLib.safeCastTo128()`

Arcadia: Addressed in PR 109 and PR 99.

Renascence: The two instances that were recommended to be fixed are fixed. An additional `safeCastTo128()` has been added in PR 109 to cast the result of `totalRealisedLiquidity - badDebt` (Link). However, the cast has been removed again as a result of the fixes for L-8. This is safe since the safe cast is not needed in this instance.

There can in theory be an edge case where `totalRealisedLiquidity < badDebt`. This would make it impossible to settle the auction with this flow that contains the calculation. So one needs to manually repay some debt for the calculation to no longer revert. This doesn't need to be fixed since it's an extreme edge case and can be resolved manually.

5 Centralization Risks

Users interacting with the Arcadia V2 protocol must be aware of the privileged roles in the protocol and which actions these privileged roles can perform. For the base-layer accounts-v2 protocol, there exists the guardian role, which is set by the owner. It can pause account deposits and withdrawals for 30 days. After 30 days, anyone can unpause the functionality for 2 days. Then, the guardian can pause the functionality again for 30 days. In addition, if oracles are decommissioned, the owner of the oracle module can update the oracles, which could obviously break the whole protocol due to incorrect pricing. Or, more simply, cause a DoS if no new Oracle is set. The accounts-v2 protocol aims to minimize trust assumptions while still granting governance the ability to intervene in an emergency. On the other hand, Creditors, which lending-v2 is one instance of, must generally be fully trusted. They have full control over the assets within a user's account. So, if a user sets a creditor for his account, it must be assessed individually. For the lending-v2 creditor, three trusted roles exist: owner, riskManager, and accountRecipient. The riskManager is set by the owner, and in turn the accountRecipient is set by the riskManager. These roles must be fully trusted. In particular, for the owner and riskManager there exist paths to effectively steal assets from accounts, e.g., by causing immediate liquidations. Lastly, any underlying protocols (collateral assets, oracles) must also be trusted.