Arcadia Security Review

Pashov Audit Group

Conducted by: Ruhum, carrotsmuggler, Todorov

January 8th 2023 - January 26th 2023

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

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

3. Introduction

A time-boxed security review of the **lending-v2** and **accounts-v2** repositories was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Arcadia

Arcadia is a non-custodial platform dedicated to improving and simplifying on-chain asset management. The V2 of the protocol enables users to construct and rebalance complex portfolios with just a single on-chain transaction, removing most of DeFi’s complexities without hiding the risks.

5. Risk Classification

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

5.1. Impact

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

5.2. Likelihood

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

5.3. Action required for severity levels

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

6. Security Assessment Summary

***review commit hashes* - 239da7a6a1f1f395a506d7e4258b29dd9d32f6ce and cd5db77bbcc561c7e8c0486311c95398b9e0e2bc**

***fixes review commit hashes* - 326587baba2f8f3ea166c9609e18171ec07b3996 and 33c11ae03ccc2066f4c80b5e078f247fdbbae92c**

Scope

The following smart contracts were in scope of the audit:

For `lending-v2`:

- `Liquidator`
- `DebtToken`
- `LendingPool`
- `Tranche`
- `guardians/LendingPoolGuardian`

For `accounts-v2`:

- `guardians/FactoryGuardian`
- `guardians/BaseGuardian`
- `guardians/RegistryGuardian`
- `Factory`
- `abstracts/Creditor`
- `Registry`
- `Proxy`
- `accounts/AccountV1`
- `accounts/AccountStorageV1`
- `oracle-modules/ChainlinkOracleModule`
- `oracle-modules/AbstractOracleModule`
- `oracle-modules/AbstractAssetModule`
- `asset-modules/AbstractAssetModule`
- `asset-modules/UniswapV3/UniswapV3AssetModule`
- `asset-modules/AbstractDerivedAssetModule`
- `asset-modules/AbstractPrimaryAssetModule`
- `asset-modules/StandardERC20AssetModule`

7. Executive Summary

Over the course of the security review, Ruhum, carrotsmuggler, Todorov engaged with Arcadia to review Arcadia. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Arcadia
Date	January 8th 2023 - January 26th 2023
Protocol Type	decentralized margin protocol

Findings Count

Severity	**Amount**
High	1
Medium	7
Low	4
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[H-01]	approvedCreditor not reset after account transfer	High	Resolved
[M-01]	Contract lacks graceful retirement feature	Medium	Resolved
[M-02]	Tranche share ratios can be manipulated by donating via liquidations	Medium	Resolved
[M-03]	Liquidators can earn extra yield by flash depositing into tranches	Medium	Resolved
[M-04]	Pending interests not processed when updating treasury weight	Medium	Resolved
[M-05]	Supporting fee-on-transfer tokens can lead to bad debt	Medium	Acknowledged
[M-06]	First Tranche depositor can withdraw all the previously earned interest	Medium	Resolved
[M-07]	Borrower pays interest on their debt while liquidation is running	Medium	Acknowledged
[L-01]	PUSH0 not supported in all chains	Low	Resolved
[L-02]	Accounts can be sold on secondary markets while liquidations are ongoing	Low	Resolved
[L-03]	Deposits / withdrawals break if all tranche weights are 0	Low	Resolved
[L-04]	Accounts can keep taking more debt during ongoing auction	Low	Resolved

8. Findings

8.1. High Findings

[H-01] `approvedCreditor` not reset after account transfer

Severity

Impact: High, user can steal funds from liquidated account

Likelihood: Medium, requires either a bad debt account, or L2 to be offline for some time

Description

The Arcadia accounts support a secondary backup creditor which is stored in the `approvedCreditor` storage variable, and can be set by the owner. The issue is that when an account is transferred, this variable is not reset, and the old owner essentially can have a backdoor to the system.

The `approvedCreditor` can call the function `flashActionByCreditor` and change the current creditor of the account. This function also calls the `_withdraw` function with some `actionTarget`, and can essentially transfer all assets in the account to that target, provided there are no outstanding loans from the old creditor. So the `approvedCreditor` can basically act like a new creditor and instead empty all the assets in an account.

The protocol prevents users from forcing this upon buyers of positions on secondary markets by requiring a minimum time elapsed between the setting of the `approvedCreditor` and a transfer of the account, and the buyers are expected to be aware of this when buying an account. However, account transfers also happen during liquidations when the lending pool internalizes a bad loan via the `auctionBoughtIn` function.

```
function auctionBoughtIn(
    address recipient
) external onlyLiquidator nonReentrant {
    _transferOwnership(recipient);
}
```

This transfers the account to the lendingPool and needs to be manually liquidated later. But due to the backdoor via `approvedCreditor`, users can steal the assets present in this account.

The `auctionBoughtIn` function is called during liquidation when the liquidation window has passed.

```
else if (block.timestamp > auctionInformation_.cutoffTimeStamp) {
ILendingPool(creditor).settleLiquidationUnhappyFlow(
    account,
    startDebt,
    msg.sender
);
IAccount(account).auctionBoughtIn(
    creditorToAccountRecipient[creditor]
);
```

This can happen if liquidators are not interested in the assets being held in the account, or if the L2 sequencer has been down for some time, preventing liquidations from happening. Since most liquidations happen via MEV bots, during times of high traffic and gas fees MEV bots might not be interested in liquidating small accounts as well, letting the timer hit the cutoff timestamp.

In this case, instead of a safe manual liquidation, the assets can be stolen via the backdoor in the account.

Recommendations

Either reset the `approvedCreditor` variable when transferring an account, or have a per-owner based `approvedCreditor`, like handled in `isAssetManager`.

8.2. Medium Findings

[M-01] Contract lacks graceful retirement feature

Severity

Impact: Medium, Users can still borrow assets from pools deemed too unsafe or unstable

Likelihood: Medium, retirements of pools are common on other platforms

Description

Commonly in lending platforms, when a certain token or lending pool has been deemed to be too risky or have been hacked, it is retired. This means that all deposits and borrows from the pool are stopped, however the pool is still kept in the system for users to withdraw and repay their loans. This is a common feature in lending platforms and is a good way to deleverage the system from risky assets while preventing users from being locked out of their funds.

This has happened multiple times in Aave, with some other examples below:

- GHST borrowing disabled on polygon
- agEUR borrowing disabled on polygon
- UST disabled on Venus protocol on BSC
- SXP disabled on Venus protocol on BSC
- TRXOld disabled on Venus protocol on BSC

The main idea being that protocols often delist tokens, and they do that by disabling deposits and borrows, but still allowing users to withdraw and repay their loans.

The issue is that this functionality does not exist in this protocol.

In the current protocol, each operation of borrow, deposit, repay and withdraw are controlled by the variables `repayPaused`, `withdrawPaused`, `borrowPaused`, `depositPaused`. These are defined and controlled in the

`LendingPoolGuardian.sol` contract. However, there is an auto-unlock feature which can trigger after a cooldown period.

```
function unpause() external override afterCoolDownOf(30 days) {
    emit PauseFlagsUpdated(
        repayPaused = false,
        withdrawPaused = false,
        borrowPaused = false,
        depositPaused = false,
        liquidationPaused = false
    );
}
```

Thus while the protocol can pause a pool to only allow withdrawals and repays they can only do so for a period of 30 days. After that, the pool is automatically unpaused and users can deposit and borrow from the pool again.

The only other way to suspend deposits is by manipulating the `lock` variable in the Tranche contracts. However, that will also prevent withdrawals.

So there is no way for the protocol to selectively disable only deposits and borrows for a pool for a long period of time. This issue can be ignored if it is a design decision by the team, however, it is being flagged here since it is a common feature in other lending protocols.

Recommendations

Have a `retire` option in the lending pool guardian which disables only deposits and borrows and cannot be unlocked after a cooldown period.

[M-02] Tranche share ratios can be manipulated by donating via liquidations

Severity

Impact: High, share ratio manipulation of ERC4626 vaults

Likelihood: Low, Requires precise timing and other conditions

Description

The contracts use the solmate ERC4626 contracts in order to prevent the share ratio manipulation attacks on the various vaults.

Here's a brief summary of the share ratio manipulation attack, useful for understanding the attack vector in the current context:

1. Alice deposits 1 wei of tokens into a manipulatable vault, which mints her 1 wei of vault tokens
2. Alice then "donates" 1e18 tokens to the vault via a direct transfer.
3. Bob now deposits 2e18 tokens into the vault. The vault calculated Bob's shares as (2e18/(1e18+1) = 1) wei of share tokens. Due to the rounding error, Bob loses access to half his tokens.

The solmate library attempts to prevent this attack by blocking the `donation` of the tokens. Normally, the total assets are calculated via the `ERC20.balanceOf` function. However, in solmate, it's kept track of in a storage variable. This makes the `donation` harder since a direct transfer of tokens will not update the total assets.

However, if there is a way to still `donate` tokens, i.e. pump up the `totalAssets` without affecting the `totalShares`, the share ratio can still be manipulated.

The liquidation mechanism in the contracts credits tokens to the owner of accounts if there is a surplus.

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

If the tranche is made to be the owner of such an account, which can be done via a simple transfer, then the `realisedLiquidityOf` of the tranche can be pumped up without minting any shares. The tranche also tracks its own assets via the functions below.

```
function totalAssetsAndSync() public returns (uint256 assets) {
    assets = LENDING_POOL.liquidityOfAndSync(address(this));
}

//@audit inside lending pool
function liquidityOfAndSync(
    address owner_
) external returns (uint256 assets) {
    _syncInterests();
    assets = realisedLiquidityOf[owner_];
}
```

So pumping up `realisedLiquidityOf` of a tranche via liquidations can increase the assets in the account without increasing shares. This can be used to manipulate the share ratio.

An attack can be done in the following steps. We assume that the tranche is empty.

1. Alice deposits 1 wei in the tranche, and gets minted 1 wei of share tokens.
2. Alice creates an account and borrows until the account is almost liquidatable.
3. Alice then transfers the nearly liquidatable account to the tranche's ownership, waits for the price to fluctuate and calls liquidate.
4. The tranche now gets credited `1e18` tokens of `surplus` due to being the `owner` of the account.
5. Now the tranche has `totalShares` = 1 wei, and `totalAssets` = 1e18+1 wei
6. Bob deposits tokens and gets rekt like previously.

Attackers can also prep tranch accounts even before deployment. Attackers can predict the address of future tranche addresses, since it is very likely that tranches will be deployed by the same protocol wallet in the future. Thus attackers can call liquidations and pre-fund future tranche accounts with this surplus. Then when a tranch finally gets added, it can have very large share ratios from the very beginning, preventing small deposits from being made due to rounding errors.

Recommendations

Since there is a way to `donate` tokens, the share ratio manipulation attack is still possible. This can be fixed via multiple ways:

1. Tranches can issue 1e10 "ghost" shares to the zero address or to the factory itself. This will prevent such share manipulation, and is the approach taken by Yieldbox and Openzeppelin's latest ERC4646 implementation.
2. OR, block the `donation` feature. This can also be done in multiple possible ways:
    1. Pay out the surplus tokens to the account owner instead of storing it in `realisedLiquidityOf`. This will not pump up the `totalAssets` count and thus prevent the `donation`.
    2. OR, put in a series of blacklists to attempt to prevent tranches from being owners of accounts. This is not a good solution since it is not future proof, and will lead to lots of code bloat.

Due to the simple nature, the main recommendation is to either mint ghost shares, or burn the first 1e10 initial minted shares.

Note: The inability for solmate-style asset tracking to prevent share ratio exploits has been highlighted in the recent Wise Lending hack, and has been commented on by alcueca here, a contributor to the original ERC4626 EIP. The current recommendation is to burn initial amounts of tokens, as referenced here or follow the latest openzeppelin or yieldbox implementations, which effectively does the same thing.

[M-03] Liquidators can earn extra yield by flash depositing into tranches

Severity

Impact: Medium, Liquidation yields can be denied to liquidity providers

Likelihood: Medium, Not applicable in current deployment configuration, but valid in general

Description

The `liquidationPenalty` part of the liquidation incentives is dealt out to the tranches as extra yield according to their weights via the `_syncLiquidationFeeToLiquidityProviders` function. This increases the amount corresponding to each tranch share. The issue is that liquidators can snipe this extra yield at no extra cost via flash deposits.

During liquidations, only the juinor-most tranche is locked, while other tranches are open and can be deposited into. The `liquidationPenalty` is given at an instant of time, thus users can theoretically deposit a very large amount to these tranche pools, collect the yield, and then withdraw it all out and pay off the flash loan.

However, the liquidators themselves can do this at no cost. So when a liquidator sees a profitable liquidation position, they can flash deposit into the tranches in the very same transaction, carry out the liquidation, and collect the termination fee, a large part of the liquidation penalty, and any discounts on the collateral price. This is a very profitable attack, and can be carried out by any liquidator.

This affects all the unlocked tranches, i.e. all except the junior-most tranche. This attack requires no frontrunning, so can be executed on all chains, irrespective of the visibility of transactions. Yield from liquidations can be denied to liquidity providers at no extra cost for the attacker.

Recommendations

Forbid deposits and withdrawals to a tranche in the same transaction / block.

[M-04] Pending interests not processed when updating treasury weight

Severity

Impact: Medium, Can lead to loss of yield for the treasury or pool participants

Likelihood: Medium, loss only happens between the weight update and next deposit/withdrawal

Description

The function `setTreasuryWeights` sets the interest weight of the treasury and also updates the global `totalInterestWeight`.

```
function setTreasuryWeights(
    uint16 interestWeightTreasury_,
    uint16 liquidationWeightTreasury_
) external onlyOwner {
    totalInterestWeight =
        totalInterestWeight -
        interestWeightTreasury +
        interestWeightTreasury_;
```

However, it is missing the `processInterests` modifier. This means that the next time `processInterests` is called the updated treasury weights will be applied from the last interest calculation point, instead of being applied from the point of the weight update.

Recommendations

Add the `processInterests` modifier to the `setTreasuryWeights` function.

[M-05] Supporting fee-on-transfer tokens can lead to bad debt

Severity

Impact: High, FOT tokens will incur bad debt

Likelihood: Low, FOT tokens are rare

Description

The protocol claims it can support fee-on-transfer (FOT) tokens if necessary. According to the pre-audit questionlist,

```
- If enabled, the collateral and liquidation factors
- (haircut on the value) will be adjusted to accommodate for the value loss due to fee
```

The issue is that changing the liquidation and collateral factors will not be enough to resolve the issue. This can be explained by a simple example.

Say Token A is a FOT token, with a 1% fee.

1. Alice deposits 100,000 tokens of token A.
2. Contract receives 99,000 tokens, but credits Alice's account with the full 100,000 tokens
3. Alice now withdraws 99,000 tokens
4. Alice receives 89100 tokens after fees. Contract has 0 tokens, since it paid out all its holdings, but contract still thinks that it holds (100,000 - 99,000)=1000 tokens.
5. Alice can now take out a loan against those 1000 tokens which the contract thinks it holds. This is entirely bad debt.

With deposits and successive withdrawals, FOT tokens can be used to create bad debt positions at very minimal costs. Since this directly affects the health of the system, this is a problem.

Recommendations

FOT tokens should always be used with post-transfer accounting logic. Since the protocol doesn't have that, it is recommended not to support FOT tokens at

all, despite the claim in the Questionlist.

[M-06] First Tranche depositor can withdraw all the previously earned interest

Severity

Impact: Medium, because it only affects the initially earned interest of the tranche.

Likelihood: Medium, because the issue is only present if there's a larger gap between the time in which the tranche starts to earn interest and time of the first deposit.

Description

When a Tranche is added to the LendingPool it's eligible to earn interest:

```
/**

            * @notice Syncs interest payments to the liquidity providers and the treasu

            * @param assets The total amount of underlying assets to be paid out as int

     * that goes to its liquidity providers.
     */
    function _syncInterestsToLiquidityProviders(uint256 assets) internal {
        uint256 remainingAssets = assets;

        uint256 trancheShare;
        uint24 totalInterestWeight_ = totalInterestWeight;
        uint256 trancheLength = tranches.length;
        for (uint256 i; i < trancheLength; ++i) {
            trancheShare = assets.mulDivDown
              (interestWeightTranches[i], totalInterestWeight_);
            unchecked {
                realisedLiquidityOf[tranches[i]] += trancheShare;
                remainingAssets -= trancheShare;
            }
        }
        unchecked {
            totalRealisedLiquidity = SafeCastLib.safeCastTo128
              (totalRealisedLiquidity + assets);

            // Add the remainingAssets to the treasury balance.
            realisedLiquidityOf[treasury] += remainingAssets;
        }
    }
```

After the first user deposits funds, the Tranche's `totalAssets` will be the deposit + the previously earned interest.

```
/**
            * @notice Returns the total amount of underlying assets, to which liquidity
     * @return assets The total amount of underlying assets.
     * @dev Modification of totalAssets() where interests are realised
       (state modification).
     */
    function totalAssetsAndSync() public returns (uint256 assets) {
        assets = LENDING_POOL.liquidityOfAndSync(address(this));
    }
```

Because the depositor holds all of the Tranche's shares, they can withdraw the whole amount by burning their tranche shares:

```
/**
     * @notice Modification of the standard ERC-4626 redeem implementation.
     * @param shares The amount of shares being redeemed.

            * @param receiver The address of the receiver of the underlying ERC20 token
     * @param owner_ The address of the owner of the shares being redeemed.
     * @return assets The corresponding amount of assets withdrawn.
     */
    function redeem(uint256 shares, address receiver, address owner_)
        public
        override
        notLocked
        notDuringAuction
        returns (uint256 assets)
    {
        if (msg.sender != owner_) {
            // Saves gas for limited approvals.
            uint256 allowed = allowance[owner_][msg.sender];

            if (allowed != type
              (uint256).max) allowance[owner_][msg.sender] = allowed - shares;
        }

        // Check for rounding error since we round down in previewRedeem.
        if ((assets = previewRedeemAndSync
          (shares)) == 0) revert TrancheErrors.ZeroAssets();

        _burn(owner_, shares);

        LENDING_POOL.withdrawFromLendingPool(assets, receiver);

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

Recommendations

Tranche shouldn't earn any interest if it didn't provide any liquidity to the creditor, i.e. check whether `realisedLiquidityOf[tranche] != 0` in

```
/**
            * @notice Syncs interest payments to the liquidity providers and the treasu
            * @param assets The total amount of underlying assets to be paid out as int
     * that goes to its liquidity providers.
     */
    function _syncInterestsToLiquidityProviders(uint256 assets) internal {
        uint256 remainingAssets = assets;

        uint256 trancheShare;
        uint24 totalInterestWeight_ = totalInterestWeight;
        uint256 trancheLength = tranches.length;
        for (uint256 i; i < trancheLength; ++i) {
            trancheShare = assets.mulDivDown
              (interestWeightTranches[i], totalInterestWeight_);
            unchecked {
                realisedLiquidityOf[tranches[i]] += trancheShare;
                remainingAssets -= trancheShare;
            }
        }
        unchecked {
            totalRealisedLiquidity = SafeCastLib.safeCastTo128
              (totalRealisedLiquidity + assets);

            // Add the remainingAssets to the treasury balance.
            realisedLiquidityOf[treasury] += remainingAssets;
        }
    }
```

[M-07] Borrower pays interest on their debt while liquidation is running

Severity

Impact: Medium, because the additional interest is limited by the auction cut off time.

Likelihood: Medium, only an issue if the user is fully liquidated.

Description

The LendingPool doesn't pause interest payments while a user's position is liquidated. When the liquidation is initiated, the user's debt position is increased to cover the initiation + termination reward as well as the liquidation penalty.

```
function startLiquidation(address initiator, uint256 minimumMargin_)
        external
        override
        whenLiquidationNotPaused
        processInterests
        returns (uint256 startDebt)
    {
        // Only Accounts can have debt, and debtTokens are non-transferrable.
        // Hence by checking that the balance of the msg.sender is not 0,
        // we know that the sender is indeed an Account and has debt.
        startDebt = maxWithdraw(msg.sender);
        if (startDebt == 0) revert LendingPoolErrors.IsNotAnAccountWithDebt();

        // Calculate liquidation incentives which have to be paid by the Account
        // owner and are minted
        // as extra debt to the Account.
        (
          uint256initiationReward,
          uint256terminationReward,
          uint256liquidationPenalty
        ) =
            _calculateRewards(startDebt, minimumMargin_);

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

        // Increase the realised liquidity for the initiator.
        // The other incentives will only be added as realised liquidity for the
        // respective actors
        // after the auction is finished.
        realisedLiquidityOf[initiator] += initiationReward;
        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 && tranches.length > 0) {
            unchecked {
                ITranche(tranches[tranches.length - 1]).setAuctionInProgress
                  (true);
            }
        }

        unchecked {
            ++auctionsInProgress;
        }

        // Emit event
        emit AuctionStarted(msg.sender, address(this), uint128(startDebt));
    }
```

The user still holds lending pool shares representing their share of the total debt. While the auction is running, interest is paid by all the debt token holders through the following modifier:

```
/**
     * @notice Syncs interest to LPs and treasury and updates the interest rate.
     */
    modifier processInterests() {
        _syncInterests();
        _;
        // _updateInterestRate() modifies the state
        //(effect), but can safely be called after interactions.
        // Cannot be exploited by re-entrancy attack.
        _updateInterestRate(realisedDebt, totalRealisedLiquidity);
    }
```

At most, the auction can run for up to 4 hours so the user only pays up to 4 hours more interest than they should. For example, MakerDAO stops accruing interest for a borrower's position as soon as the auction is initiated to cover the liquidation.

Recommendations

Positions that are liquidated shouldn't accrue additional interest.

8.3. Low Findings

[L-01] PUSH0 not supported in all chains

The contracts use solidity version `0.8.22`, which can generate `PUSH0` opcodes depending on the evm version specified in the compiler settings. This opcode is not supported in all chains, and can cause the contracts to fail on deployment.

There is no `evm_version` specified in the foundry.toml file, so the contracts will be compiled with the default evm version, which is `paris`, which doesn't generate `PUSH0` opcodes, and thus is fine for now. However if this ever changes to `shanghai` or later, contract deployments can start failing on chains that do not support `PUSH0`.

It is recommended to either roll back the solidity version, OR hardcode the evm version to `paris` in the foundry.toml file to prevent any unforeseen changes in the configuration.

[L-02] Accounts can be sold on secondary markets while liquidations are ongoing

Accounts can be transferred via the `safeTransferFrom` function which calls the `transferOwnership` on the account. This `transferOwnership` does not have the `notDuringAuction` modifier, and can thus carry out transfers while an auction is going on, as long as the cooldown period has passed. Thus buyers of accounts on secondary markets can be scammed, since accounts in an auction can enter a healthy state after a partial liquidation, or due to price fluctuations, thus an `isAccountUnhealthy` check is not sufficient for the buyer to be aware. The buyer can then lose all the collateral assets after a successful bid by some other liquidator after the transfer.

Consider adding the `notDuringAuction` modifier to the `transferOwnership` function

[L-03] Deposits / withdrawals break if all tranche weights are 0

The `addTranche` function has no zero-value check in the LendingPool.sol contract. But if all added tranches have a weight of 0, and thus if `totalInterestWeight` amounts to 0, the interest calculations can break due to a division by 0 in the `_syncInterestsToLiquidityProviders` function.

```
trancheShare = assets.mulDivDown(
                interestWeightTranches[i],
                totalInterestWeight_
            );
```

Consider adding a zero value check in both the `addTranche` and `setTrancheWeights` functions, for both `interestWeightTranches` and `liquidationWeightTranches` variables.

[L-04] Accounts can keep taking more debt during ongoing auction

The function `borrow` in the lendingPool contract can be used to take out more debt against the account's collateral. This function mints debt tokens to the account and then does a health check.

```
// Mint debt tokens to the Account.
_deposit(amountWithFee, account);

// Add origination fee to the treasury.
unchecked {
    if (amountWithFee - amount > 0) {
        totalRealisedLiquidity = SafeCastLib.safeCastTo128(
            amountWithFee + totalRealisedLiquidity - amount
        );
        realisedLiquidityOf[treasury] += amountWithFee - amount;
    }
}

// UpdateOpenPosition checks that the Account indeed has opened a margin account
// for this Lending Pool and
// checks that it is still healthy after the debt is increased with
// amountWithFee.
// Reverts in Account if one of the checks fails.
uint256 accountVersion = IAccount(account).increaseOpenPosition(
    maxWithdraw(account)
);
```

The health check is done in the `increaseOpenPosition` function in the `Account` contract.

```
function increaseOpenPosition(
    uint256 openPosition
)
    external
    onlyCreditor
    nonReentrant
    updateActionTimestamp
    returns (uint256 accountVersion)
{
    // If the open position is 0, the Account is always healthy.
    // An Account is unhealthy if the collateral value is smaller than the used
    // margin.
    // The used margin equals the sum of the given open position and the minimum
    // margin.
    if (
        openPosition > 0 &&
        getCollateralValue() < openPosition + minimumMargin
    ) {
        revert AccountErrors.AccountUnhealthy();
    }

    accountVersion = ACCOUNT_VERSION;
}
```

This function does a health check by checking if the used margin is smaller than the value of the collateral, however, the function does not check if there is an ongoing auction. This means an account can take on more debt as long as they fulfill the health check.

The issue is that during liquidation, accounts can become healthy after a partial liquidation. This is because liquidations start with a high multiplier, and some user might be willing to take that deal in order to secure a prized ERC721 from the asset list, and push the account to a healthy state again. However then the account owner can take more debt and worsen the health factor of the system. If the price fluctuates, then the system can be pushed into a bad debt state due to this action.

Since the account owner can actively take out more debt during an auction, this should be fixed.

Consider adding the `notDuringAuction` modifier to the `increaseOpenPosition` function in the `Account` contract.