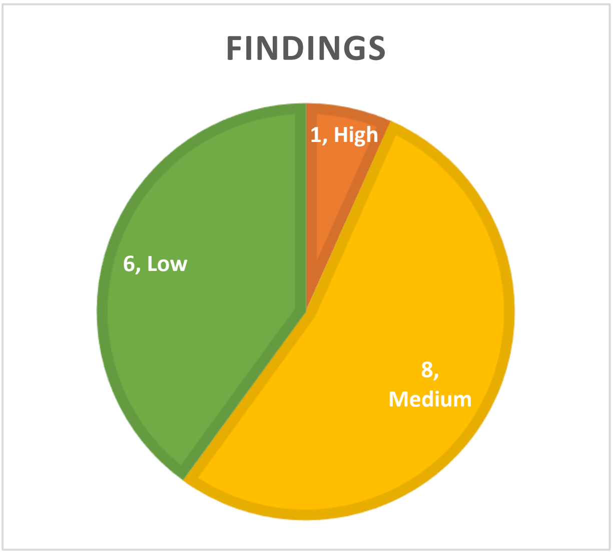
Trust Security

Smart Contract Audit

Arcadia Finance v2

25/12/2023

Executive summary

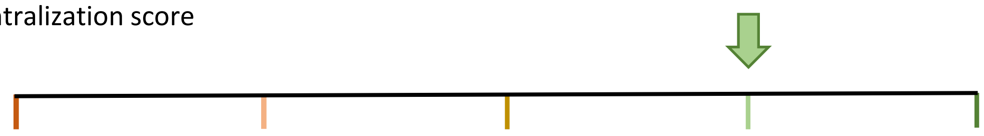


Category	Margin Accounts
Audited file count	23
Lines of Code	3075
Auditor	Lambda, 0xladboy
Time period	27/11/2023 – 13/12/2023

Findings

Severity	Total	Fixed	Acknowledged
High	1	1	0
Medium	8	7	1
Low	6	4	2

Centralization score



Centralized Decentralized

Signature

Digitally signed by Trust
Date: 2023.12.25 16:17:39 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	13/12/2023	Client report
0.2	25/12/2023	Mitigation review

Contact

Trust
trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- accounts/AccountV1.sol
- Registry.sol
- asset-modules/AbstractDerivedAssetModule.sol
- asset -modules/UniswapV3/UniswapV3AssetModule.sol
- asset-modules/AbstractPrimaryAssetModule.sol
- Factory.sol
- asset-modules/AbstractAssetModule.sol
- oracle-modules/ChainlinkOracleModule.sol
- libraries/BitPackingLib.sol
- Proxy.sol
- asset-modules/StandardERC20AssetModule. sol
- libraries/AssetValuationLib.sol
- guardians/RegistryGuardian.sol
- abstracts/Creditor.sol
- guardians/BaseGuardian.sol
- guardians/FactoryGuardian.sol
- accounts/AccountStorageV1.sol
- oracle-modules/AbstractOracleModule.sol
- LendingPool.sol
- Liquidator.sol
- Tranche.sol
- guardians/LendingPoolGuardian.sol
- DebtToken.sol

Repository details

- **Repository URL:** https://github.com/arcadia-finance/accounts-v2
- **Commit hash:** 4f4057d4ebafb0fbf739cbf627ff397d0e569c19
- **Repository URL:** https://github.com/arcadia-finance/lending-v2
- **Commit Hash:** 03f930c55094582cbb11ff25791a3e560bfa917c

The mitigation review has been completed on a per-PR basis. The audited change can be found on each issue separately.

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

Lambda is a Security Researcher and Developer with multiple years of experience in IT security and traditional finance. This experience combined with his academic background in Data Science, Mathematical Finance, and High-Performance Computing enables him to thoroughly examine even the most complicated code bases, resulting in several top placements in various audit contests.

0xladboy is a researcher specializing in blockchain security, known for consistently providing top-tier audits for clients. In 2023, he was recognized among the top 10 in the Code4rena rankings, and he is the top 5 in the past 90 days (about 3 months) in the code4rena leaderboard at the time of the audit.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.
Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project kept code as simple as possible, reducing attack risks
Documentation	**Good**	Project is mostly very well documented.
Best practices	**Good**	Project consistently adheres to industry standards.
Centralization risks	**Good**	Project is mostly decentralized.

Findings

High severity findings

TRST-H-1: Lack of input validation in account parameter allows user to trigger liquidation flow and steal funds

- **Category:** Logical flaws
- **Source:** Liquidator.sol, LendingPool.sol
- **Status:** Fixed

Bug description

The function *liquidateAccount()* is used to initiate the liquidation of an account. It accepts the argument **account** but does not validate that this is a valid account that was created by the factory. An attacker can abuse this to pass in a malicious address that allows them to immediately settle the auction and steal funds.

To exploit the issue, the attacker creates a contract that adheres to the Account interface but lets them control all returned values. This account returns a valid creditor (the lending pool), and an arbitrary debt amount. An excerpt of such a contract could look like this:

```
    function setCreditor(address creditor) public {
        require(msg.sender == owner);
        address creditor = _creditor;
    }

    function setDebtAmount(uint256 _debt) public {
        require(msg.sender == owner);
        uint256 debt = _debt;
    }

    function startLiquidation(address initiator)
        external
        returns (
            address[] memory,
            uint256[] memory,
            uint256[] memory,
            address,
            uint256,
            AssetValueAndRiskFactors[] memory
        ); {

            address[] memory assetAddresses,
            uint256[] memory assetIds,
            uint256[] memory assetAmounts,

            AssetValueAndRiskFactors[] memory assetValues
            return (
                assetAddresses,
                assetIds,
                assetAmounts,
                creditor, // valid lending pool from malicious account
                debt, // this can be manipulated by the attacker.
            )
        }
```

Directly after the initiation of the liquidation, the attacker calls *endAuction()* and provides the same contract address there. The function now queries the collateral value and margin of the account, but these values can be controlled completely by the attacker. They set them such that the collateral value is larger than the used margin, which will trigger the happy flow:

```
        (uint256 initiationReward, uint256 terminationReward, uint256
liquidationPenalty) = _calculateRewards(startDebt);

        // Pay out the "liquidationPenalty" to the LPs and Treasury.
        _syncLiquidationFeeToLiquidityProviders(liquidationPenalty);

        // Unsafe cast: sum will revert if it overflows.
        totalRealisedLiquidity = uint128(totalRealisedLiquidity +
terminationReward + liquidationPenalty + 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;
        }
```

This attack can be repeated an arbitrary number of times (or with arbitrary high values for **startDebt**) to drain the liquidity pool.

Recommended mitigation

We recommend validating the **account** parameter. It should be ensured that the account that is subject to liquidation was created by the factory and is therefore a non-malicious implementation.

Team response

Fixed in PR 89.

Mitigation review

The fix ensures that the account address is deployed by the factory contract.

Medium severity findings

TRST-M-1: Auction cannot be settled in unhappy flow if some collateral token in the account cannot be transferred after liquidation cutoff time

- **Category:** Logical flaws
- **Source:** AccountV1.sol
- **Status:** Fixed

Description

If the debt is not fully repaid after the liquidation cutoff time, the liquidation should be settled in the unhappy flow. In this flow, the remaining collateral assets are transferred to the recipient address set by the risk manager, which will sell them to cover the debt.

This is done in the function *auctionBoughtIn()*:

```
    function auctionBoughtIn(address recipient) external onlyLiquidator {
        (address[] memory assetAddresses, uint256[] memory assetIds,
uint256[] memory assetAmounts) =
            generateAssetData();
        _withdraw(assetAddresses, assetIds, assetAmounts, recipient);
    }
```

However, there can be scenarios where some of these transfers will fail. For instance, the system might use a rebasing token (where the balance decreased), the token may be temporarily paused, or the recipient may be blocked. In such scenarios, settling the auction will not be possible and the recipient will not be able to seize the rest of the assets.

Note that this issue was found and reported by the Arcadia team during the audit.

Recommended mitigation

We recommend making the *auctionBoughtIn()* flow as simple as possible by just transferring the account to the recipient address instead of sending all assets to them.

Team response

Fixed in PR 123.

Mitigation review

The fix transfers the ownership of the account to the liquidator such that the unhappy flow will never be blocked.

TRST-M-2: Account owner can preserve account control by setting a malicious creditor address and liquidator address after account ownership is transferred

- **Category:** Validation issues
- **Source:** AccountV1.sol
- **Status:** Fixed

Description

Arcadia mints an NFT for every account that is created by the factory. These NFTs can be transferred and users can therefore trade them on secondary markets.

```
     * @notice Function used to transfer an Account between users based on
Account id.
     * @param from The sender.
     * @param to The target.
     * @param id The id of the Account that is about to be transferred.
     * @dev This method overwrites the transferFrom function in ERC721.sol
to
     * also transfer the Account proxy contract to the new owner.
     */
    function transferFrom(address from, address to, uint256 id) public
override {
        IAccount(allAccounts[id - 1]).transferOwnership(to);
        super.transferFrom(from, to, id);
    }
```

However, before doing so, a user can open a margin account and set a malicious **creditor**, as this argument is not validated:

```
    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 **creditor** address has extended privileges and controls the address of the liquidator. Moreover, they can perform flash actions, which can be used to withdraw assets from the account (if it is still healthy afterwards).

A malicious user can therefore sell an account but keep control of it by changing the creditor address to a contract that they control. Note that this could also be done just before the transfer (by frontrunning it), so validating that the creditor is non-malicious before a purchase is insufficient.

Recommended mitigation

We recommend whitelisting the creditor addresses to ensure that there can be no valid accounts with a malicious creditor.

Team response

Fixed in PR 122.

Mitigation review

The fix ensures that no transfer can happen directly after an *openMarginAccount()* call (with a cooldown period of 5 minutes). The receiver/buyer therefore still has to validate that the creditor is non-malicious before a transfer, but they can be sure that it will not be changed directly before the transfer.

TRST-M-3: Invalid oracle price can be consumed

- **Category:** Oracle issues
- **Source:** ChainlinkOracleModule.sol
- **Status:** Fixed

Description

Arcadia uses oracle prices to determine the current value of all assets, making them an important part of the overall system. However the system may use invalid prices in some scenarios.

This can occur because the function *getRate()* queries the chainlink oracle and returns the price directly without validating if the returned price is fresh.

```
      function getRate(uint256 oracleId) external view override returns
(uint256 oracleRate) {
        OracleInformation memory oracleInformation_ =
oracleInformation[oracleId];

        // If the oracle is not active (decommissioned), the transactions
revert.
        // This implies that no new credit can be taken against assets that
use the decommissioned oracle,
        // but at the same time positions with these assets cannot be
liquidated.
        // A new oracleSequence for these assets must be set ASAP in their
Asset Modules by the protocol owner.
        if (!oracleInformation_.isActive) revert InactiveOracle();

        (, int256 tempRate,,,) =
IChainLinkData(oracleInformation_.oracle).latestRoundData();

        // Only overflows at absurdly large rates, when rate >
type(uint256).max / 10 ** (18 - decimals).
        // This is 1.1579209e+59 for an oracle with 0 decimals.
        unchecked {
            oracleRate = uint256(tempRate) *
oracleInformation_.unitCorrection;
        }
    }
```

While there is a function *decommissionOracle()* that can be used to decommission an oracle that returns an invalid price, it has to be called explicitly and this validation is not done automatically before consuming prices. Therefore, if no one calls this function, the system will continue to consume invalid prices. This can for instance lead to scenarios where stale prices are consumed.

Recommended mitigation

We recommend validating the oracle answer whenever it is consumed and not only when someone explicitly requests to decommission an oracle. Moreover, consider validating the oracle more extensively. This resource can be helpful when determining the validation steps.

Team response

Fixed in PR 128.

Mitigation review

The oracle response is now validated whenever it is used, and it is checked that the sequencer is online. The grace period check for the sequencer does not work when address(0) is passed in as **creditor** because **riskParams[creditor].gracePeriod** will always be 0. To address this, a global grace period could be introduced. However, this is only a minor detail with limited impact.

TRST-M-4: Interest is still accruing when repay is paused

- **Category:** Accounting issues
- **Source:** LendingPool.sol
- **Status:** Fixed

Description

The function repay can be paused temporarily, making it impossible for users to repay their open debt.

```
function repay(uint256 amount, address account) external whenRepayNotPaused
processInterests {
```

However, *calcUnrealisedDebt()* does not consider this and interest is still accruing during that time:

```

    function calcUnrealisedDebt() public view returns (uint256
unrealisedDebt) {
        unchecked {
            //gas: Can't overflow for reasonable interest rates.
            uint256 base = 1e18 + interestRate;

            // gas: Only overflows when (block.timestamp -
lastSyncedBlockTimestamp) > 1e59
            // in practice: exponent in LogExpMath lib is limited to 130e18,
            // Corresponding to a delta of timestamps of 4099680000 (or 130
years),
            // much bigger than any realistic time difference between two
syncs.
            uint256 exponent = ((block.timestamp - lastSyncedTimestamp) *
1e18) / YEARLY_SECONDS;
```

```
            // gas: Taking an imaginary worst-case scenario with max
interest of 1000%
            // over a period of 5 years.
            // This won't overflow as long as openDebt <
3402823669209384912995114146594816
            // which is 3.4 million billion *10**18 decimals.
            unrealisedDebt = (realisedDebt * (LogExpMath.pow(base, exponent)
- 1e18)) / 1e18;
        }

        return SafeCastLib.safeCastTo128(unrealisedDebt);
    }
```

This can lead to situations where users will be forcefully liquidated and have no chance to repay their debt.

Recommended mitigation

Consider not accruing interest when the user cannot repay the open debt.

Team response

Fixed in PR 94.

Mitigation review

The interest rate is set to 0 when the repay functionality is paused, ensuring that no more interest is accrued.

TRST-M-5: Liquidations cannot be settled in some edge cases

- **Category:** Logical flaws
- **Source:** LendingPool.sol, DebtToken.sol
- **Status:** Fixed

Description

When all collateral is sold, but there is still remaining debt, *_settleAuction()* initiates settlement via the unhappy flow. *settleLiquidationUnhappyFlow()* removes any remaining debt from the account in the end:

```
        // Remove the remaining debt from the Account now that it is written
off from the liquidation incentives/Liquidity Providers.
        _withdraw(openDebt, account, account);
```

_withdraw() rounds down when calculating the shares to withdraw, but also reverts when this results in zero shares:

```
    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();
```

In edge cases with a very small remaining debt, this can therefore lead to a situation where the settlement of the auction fails.

Recommended mitigation

If there is only a very small amount of debt left that will cause a revert, no withdrawal should be initiated.

Team response

Fixed in PR 92.

Mitigation review

The logic was changed to properly handle the edge case.

TRST-M-6: Junior tranche depositors can frontrun liquidations

- **Category:** Economical attacks
- **Source:** Tranche.sol
- **Status:** Acknowledged

Description

During a liquidation, no one can withdraw (or deposit) into the most junior tranche. This makes sense because this tranche will bear any losses from the liquidation and the depositors are in turn also rewarded for this higher risk.

However, a depositor that sees the transaction to start a liquidation in the mempool can frontrun this transaction and withdraw before it starts. Like that, they get the higher interest rate without any risk. This could be done for every liquidation or the depositor could also analyze which liquidations likely will incur a loss and only frontrun those.

This is a significant risk for the protocol because it can result in situations where almost all liquidity is withdrawn from the junior tranche before a liquidation, therefore eliminating the intended safety mechanisms. Moreover, it means that the intended seniority structure of the tranches (with a different risk / payout structure) does not hold in practice.

Recommended mitigation

Do not allow immediate withdrawals. They should be announced first and the user should then be able to withdraw *x* minutes later if no auction is in progress.

Team response

Acknowledged. This will not be a problem for the release (on base with only one tranche), but will be fixed in the second tranche if and when there is a second tranche.

TRST-M-7: Initial numeraire is not validated

- **Category:** Validation issues
- **Source:** AccountV1.sol
- **Status:** Fixed

Description

Whenever the numeraire is changed within the Account contract, the following check within *_setNumeraire()* is performed to see if the registry supports it:

```
if (!IRegistry(registry).inRegistry(numeraire_)) revert
AccountErrors.NumeraireNotFound();
```

However, for the initial **numeraire** within *initialize()*, this check is not done (if no creditor is provided). When an unsupported **numeraire** is provided there, the account ends up in a state where deposits are possible (because no health check is done after a deposit), whereas withdrawals fail.

This happens because *Account.getCollateralValue()* calls *getCollateralValue()* on the registry, which will try to convert the amount into the (unsupported) numeraire:

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

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

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

This is problematic, but the state can be resolved by setting a valid numeraire (via *setNumeraire()*). However, the function cannot be used when a creditor is set:

```
    function setNumeraire(address numeraire_) external onlyOwner
nonReentrant {
        if (creditor != address(0)) revert
AccountErrors.CreditorAlreadySet();
        _setNumeraire(numeraire_);
    }
```

Moreover, *_openMarginAccount()* only changes the **numeraire** (and therefore validates it) if it is different to the current one:

```
    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_);
    }
```

In the following (relatively unlikely) scenario, the funds would therefore be stuck:

- Account is initialized with no creditor and non-supported **numeraire**
- User deposits funds
- User sets a creditor via *openMarginAccount()* that has the same non-supported **numeraire**, so no validity check is performed
- User cannot change the **numeraire** via *setNumeraire()* (because a creditor is set), but also not by providing a new creditor (because *openMarginAccount()* reverts when calling *closeMarginAccount()* on the current one):

Recommended mitigation

Validate the initial numeraire.

Team response

Fixed in PR 90.

Mitigation review

The numeraire parameter is no longer passed in when the account contract is initialized, the user has to call *setNumeraire()* explicitly now.

TRST-M-8: Locked funds in tranche when a new one is added during a liquidation

- **Category:** Logical flaws
- **Source:** LendingPool.sol
- **Status:** Fixed

Description

When a liquidation is started, LendingPool locks up the most junior tranche to prevent any deposits / withdrawals:

```
        // 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);
```

When the last auction is ended, this is undone again:

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

However, the owner can add a new tranche in the meantime:

```
    function addTranche(address tranche, uint16 interestWeight_, uint16
liquidationWeight) external onlyOwner {
        if (isTranche[tranche]) revert
LendingPoolErrors.TrancheAlreadyExists();

        totalInterestWeight += interestWeight_;
        interestWeightTranches.push(interestWeight_);
        interestWeight[tranche] = interestWeight_;

        totalLiquidationWeight += liquidationWeight;
        liquidationWeightTranches.push(liquidationWeight);

        uint8 trancheIndex = uint8(tranches.length);
        tranches.push(tranche);
        isTranche[tranche] = true;

        emit TrancheAdded(tranche, trancheIndex);
        emit TrancheWeightsUpdated(trancheIndex, interestWeight_,
liquidationWeight);
    }
```

In that case, *ITranche(tranches[tranches.length - 1]).setAuctionInProgress(false)* will be called on the newly added tranche. But it will be never called on the original junior tranche.

Therefore, deposits / withdrawals will never be possible there (irrevocably) and all funds of the tranche would be locked up.

Recommended mitigation

Do not allow adding new tranches while an auction is in progress.

Team response

Fixed in PR 91.

Mitigation review

Tranches cannot be added when an auction is in progress.

Low severity findings

TRST-L-1: The check to ensure the creditor address version is compatible with the lending pool during account upgrade be bypassed if user set creditor address to a malicious one

- **Category:** Validation issues
- **Source:** AccountV1.sol
- **Status:** Acknowledged

Description

When an account upgrade is performed, it is checked if the creditor is compatible with the new version after verifying the merkle proof that ensures that the upgrade path is supported. However, because the creditor address is user-controlled and not verified, it can happen that this contract returns true, although it is not supported (by accident or on purpose).

```
        // If a Creditor is set, new version should be compatible.
        if (creditor != address(0)) {
            (bool success,,,) =
ICreditor(creditor).openMarginAccount(newVersion);
            if (!success) revert AccountErrors.InvalidAccountVersion();
        }
```

The impact depends on the new implementation logic. In the worst case, the user could end up with an account and creditor implementation that do not adhere to the same standards.

Recommended mitigation

We recommend the protocol to whitelist and validate the creditor address to make sure the user cannot set arbitrary creditor address when opening a margin account.

Team response

Acknowledged.

TRST-L-2: The usage of transfer is not recommended

- **Category:** Gas-related flaws
- **Source:** AccountV1.sol
- **Status:** Fixed

Description

The Account contract has a *skim()* function to rescue any ETH that was sent to it. The function uses *transfer()* to transfer the ETH to the recipient. Because this limits the gas usage of the recipient to 2,300 (which may be insufficient for smart contracts), its usage is no longer recommended.

Recommended mitigation

Use *call()* instead of *transfer()* to rescue ETH.

Team response

Fixed in PR 124.

Team response

Fixed.

TRST-L-3: minAnswer / maxAnswer from Chainlink oracle is deprecated

- **Category:** Validation issues
- **Source:** ChainlinkOracleModule.sol
- **Status:** Fixed

Description

According to the Chainlink documentation, the values **minAnswer** and **maxAnswer** is are longer supported and used for Chainlink feeds:

„This value is no longer used on most Data Feeds. Evaluate if your use case for Data Feeds requires a custom circuit breaker and implement it to meet the needs of your application. See the Risk Mitigation page for more information.“

While some feeds such as MATIC / USD still return reasonable values, other feeds just set **minAnswer** to 1 and **maxAnswer** to type(int192).max, meaning that they cannot be used for an effective validation.

Recommended mitigation

As mentioned in the Chainlink documentation, we recommend implementing a custom circuit breaker logic and evaluating the usage of a backup oracle.

Team response

Fixed in PR 128.

Mitigation review

The oracle validation logic was changed and no longer uses these values.

TRST-L-4: Unnecessary factory pause restriction

- **Category:** Logical errors
- **Source:** FactoryGuardian.sol
- **Status:** Fixed

Description

The pause function of the RegistryGuardian has the modifier *afterCoolDownOf(32 days)* to ensure that the guardian can call the function (and therefore pause the registry) only every 32 days.

This is a good measure for decentralization because anyone can call *unpause()* after 30 days, so there is a guarantee for the user that the registry will not be paused indefinitely.

However, the *pause()* function of the FactoryGuardian also has this modifier. For the factory, there is no public *unpause()* function with the reasoning "No reason to be able to create an Account if the owner of the Factory did not unpause *createAccount()*."

Therefore, this cooldown is an unnecessary restriction without any benefits that could hinder the teams ability to respond to incidents in the worst case.

For instance, let's say the creation is paused by the guardian and then unpaused a few days later by the owner because the issue was resolved. If there is another issue a few days later, the guardian will not be able to pause again.

Recommended mitigation

Remove the modifier from the *pause()* function.

Team response

Fixed in PR 121.

Mitigation review

Fixed by removing the modifier.

TRST-L-5: Users can frontrun transfers of accounts to withdraw assets

- **Category:** Frontrunning attacks
- **Source:** AccountV1.sol
- **Status:** Fixed

Description

One feature of Arcadia is that accounts are an NFT that can be transferred and traded. However, assessing the value of the account is almost impossible in practice because the previous owner can frontrun transfers with withdrawals that change it.

For instance, consider the following scenario:

- Bob wants to buy Alice's account with a current surplus of 1,000 USDC. He verifies carefully that it is non malicious and pays 1,000 USD for it.
- Alice frontruns the *transfer()* of the NFT and withdraws 900 USDC. She gets 1,000 USD for the account, although it is only worth 100 USDC.

Recommended mitigation

Consider introducing a feature that locks an account temporarily. This would allow buyers to assess the account value and makes sure that it does not change before the transfer.

Team response

Fixed in PR 122.

Mitigation review

Fixed by introducing a cooldown period for transfers after certain actions.

TRST-L-6: Malicious asset manager can frontrun flash action to consume signatures

- **Category:** MEV attacks
- **Source:** AccountV1.sol

- **Status:** Acknowledged

Description

Besides direct ERC20 transfers, flash actions support the transfer of the owner's funds via signatures using Uniswap's Permit2 system. The following code within *_flashAction()* initiates these transfers:

```
        // If the function input includes a signature and non-empty token
permissions,
        // initiate a transfer from the owner to the actionTarget via
Permit2.
        if (signature.length > 0 && permit.permitted.length > 0) {
            _transferFromOwnerWithPermit(permit, signature, actionTarget);
        }
```

_transferFromOwnerWithPermit() then calls the Permit2 contract to transfer the tokens to the action target of the flash action:

```
PERMIT2.permitTransferFrom(permit, transferDetails, owner, signature);
```

An asset manager that sees these calls with a valid signature in the mempool can extract the signature and craft its own flash action with the same action target.

In the worst case, the asset manager could use this to steal the tokens from the owner. We illustrate this with the MultiCall flash action (which was out of scope for this audit): If a user uses permits in combination with this action, the asset manager can frontrun the flash action with their own flash action that just steals the token. However, because the asset manager is assumed to be a trusted role that can transfer funds from the owner over other methods (if the owner has authorized them), the impact is limited. It may be unexpected for a user that used a permit-based approval for a one-off action (and did not assume that this can also be used by the asset manager), but it requires a malicious asset manager.

Recommended mitigation

Because the flash actions were designed to be very general (with a bytes argument **actionData** that is decoded and interpreted according to the details of the action), ensuring that the frontrunning never results in lost funds is very hard without a significant redesign. We therefore recommend removing signature-based approvals for flash actions.

Team response

The team regards this function as designed and acknowledges the finding.

Additional recommendations

TRST-R-1: Should check if oracle is stale when adding new oracle

It is recommended that protocol should validate if the price oracle is not stale when a new oracle is added.

Team response

Should be covered with the *checkOracleSequence()* when setting an oracle for an address.

Mitigation review

Whenever an oracle starts getting used for an asset, this is indeed checked by *checkOracleSequence()*. Not performing the check when adding the oracle should therefore be fine, unless the team wants to avoid that stale oracles can be added to the system (even if they are not used for pricing).

TRST-R-2: Avoid hardcoding permit2 address

It is recommended that the permit2 address is not hardcoded. The address should be passed as an argument to the constructor instead.

Team response

Acknowledged.

TRST-R-3: Inconsistent naming

The codebase sometimes uses different names that refer to the same concept:

- The function *_calculateReward()* returns three parameters: **initationReward**, **terminationReward** and **liquidationReward** according to the function comments. In *startLiquidation()*, the second parameter is called **closingReward**.
- In *bid()*, the result of *_calculateBidPrice()* is stored as **price**, but then passed as the argument **amount** to *auctionRepay()*.

We recommend using the same name everywhere for consistency.

Team response

Fixed in PR 93.

Mitigation review

Fixed.

TRST-R-4: Access control for startLiquidation()

The function *startLiquidation()* does not have any access control and can be called by anyone. While there is a check for the current debt (which implicitly checks for the existence of an account), an explicit access control could be added in case the previous invariant is no longer true at some point in the future.

Team response

Acknowledged.

TRST-R-5: After liquidation starts, user's asset can still be liquidated after user deposits more collateral to make account healthy

Users are able to deposit collateral into the account at any time. If an account becomes healthy during a liquidation because the user deposited more collateral, further bids are still possible. We recommend verifying if this is the intended behavior. If so, it is recommended to document it explicitly, because users may not be aware of this.

Team response

Acknowledged.

TRST-R-6: Dust collateral logic could be improved

To avoid situations where the liquidation of an account with a lot of dust collateral positions is not profitable, any collateral with a value less than **minUsdValue** is not added to the total collateral. If a user therefore has a lot of small collateral positions, the value of the account will be much lower than the sum of the individual positions.

To avoid this, consider using an alternative logic that looks at the total account value. If this is below a threshold, the collateral could still be ignored. But this would improve the experience for users with a lot of small positions, as the sum of the positions might be above the global threshold.

Team response

Acknowledged.

TRST-R-7: Protocol admin should not update liquidation parameter when there are ongoing liquidations

The lendingPool owner can call *setLiquidationParams()* to adjust liquidation parameter at any time. This can lead to some discrepancies when these changes are performed during an auction. For instance, consider the following scenario:

1. User's account is unhealthy, liquidation starts, the initial reward, termination reward and liquidation penalty is minted as debt to the user account by calling *_calculateReward()*.
2. The owner calls *setLiquidationParams()* and increases the **maxInitiationFee** and **maxTerminationFee**.
3. When the liquidation is settled, the same amount of **startDebt** is used to compute the initial reward, termination reward and liquidation penalty again, but now with a higher termination reward. This termination reward was not minted as debt initially.

Team response

setLiquidationParams() already checks that there are no ongoing liquidations.

Mitigation review

Fixed.

TRST-R-8: Inaccurate comments

The following comments are Inaccurate

- Above the Registry contract, "*It manages the Action Multicall.*" is mentioned.
- In the description of the FactoryGuardian contract, it is stated that "Anyone to unpause all functionalities after a fixed cool-down period."

We recommend fixing these comments.

Team response

Fixed in PR 125.

Mitigation review

Fixed.

TRST-R-9: Document creditor overexposure

When assets are withdrawn, it is not checked if the exposure after the withdrawal is higher than the maximum exposure of the creditor, although withdrawals can increase the exposure.

While adding this check could prevent withdrawals in some scenarios. It is recommended to document clearly that the max exposure parameter is not strictly enforced and that the actual exposure can be higher in some cases.

Team response

Documented in PR 126.

Mitigation review

Fixed.

TRST-R-10: Hardcoded stale period as 2 days can be too long

In the current implementation of decommission oracle, if the oracle price is not updated for 2 days, it is considered stale:

```
         try IChainLinkData(oracle).latestRoundData() returns (uint80,
int256 answer, uint256, uint256 updatedAt, uint80)
        {
            if (answer <=
IChainLinkData(IChainLinkData(oracle).aggregator()).minAnswer()) {
                oracleIsInUse = false;
            } else if (answer >=
IChainLinkData(IChainLinkData(oracle).aggregator()).maxAnswer()) {
                oracleIsInUse = false;
            } else if (updatedAt <= block.timestamp - 2 days) {
                oracleIsInUse = false;
            }
        } catch {
            oracleIsInUse = false;
        }
```

Because of the volatility of some markets, a period of 2 days can be very long. Consider using a shorter or configurable period.

Team response

Fixed in PR 128.

Mitigation review

The period is now configurable per oracle.

Centralization risks

TRST-CR-1: The owner can add malicious account implementations

The owner can use *setNewAccountInfo()* to add new account implementations at any time. This does not affect existing accounts, but it would allow a malicious owner to upgrade their own account to a malicious implementation afterwards, as long as the new version is approved by the creditor (for instance when they collude with the owner or are controlled by the same private key).

Because accounts are considered to be trusted and non-malicious in Arcadia, such a malicious implementation could drain the whole liquidity in the worst case. For instance, it could simply ignore liquidations or report that it is healthy when it is not.

TRST-CR-2: Risk manager setting and configuration

When an auction did not end within the cutoff time, all remaining assets are transferred to the risk manager of a creditor and a manual liquidation is done.

This address (that is set by the owner of the LendingPool) therefore has to be trusted and could steal assets if it is malicious.

TRST-CR-3: Lending pool parameters

The owner of the lending pool can use *setLiquidationParameters()* and *setInterestParameters()* to change the liquidation and interest parameters arbitrarily.

This could be abused to set parameters which make the protocol economically insecure (for instance because nobody has an incentive to liquidate users).

Systematic risks

TRST-SR-1 Underlying token failures could impact health of all accounts

According to the developers, the protocol intends to use different tokens (COMP, USDC, and USDT) that have the potential for infinite minting by the owner.

This and other attacks on the underlying tokens could have a severe impact on the protocol, as it could result in situations where almost all accounts become liquidatable at once.

TRST-SR-2 Dependency on Chainlink price feeds

The protocol solely relies on Chainlink oracles without any backup mechanism. If these oracles start to report wrong values (that are still in the valid range), many positions could be liquidated, or it could allow users to borrow unsafe amounts. Note that Chainlink oracles have reported wrong values in the past for short timespans.