METATRUST

Draft

Security Assessment for

23-2021-08-notional (1Positive-SP) (1Negative-FLP)

July 23, 2023

Executive Summary

Overview	
Project Name	23-2021-08-notional (1Positive-SP) (1Negative-FLP)
Codebase URL	https://github.com/code-423n4/2021-08-notional
Scan Engine	AI Analyzer
Scan Time	2023/07/23 22:07:29
Commit Id	8368d59

Critical Issues	The issue can cause large economic losses, large-scale data disorder, loss of control of authority management, failure of key functions, or indirectly affect the correct operation of other smart contracts interacting with it.
High Risk Issues	The issue puts a large number of users' sensitive information at risk or is reasonably likely to lead to catastrophic impacts on clients' reputations or serious financial implications for clients and users.
Medium Risk Issues	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low Risk Issues	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational Issue	The issue does not pose an immediate risk but is relevant to security best practices or Defence in Depth.

Total	
Critical Issues	0
High risk Issues	4
Medium risk Issues	0
Low risk Issues	0
Informational Issues	0

Critical Issues	0%	0
High risk Issues	100%	4
Medium risk Issues	0%	0
Low risk Issues	0%	0
Informational Issues	0%	0

Summary of Findings

MetaScan security assessment was performed on **July 23, 2023 22:07:29** on project **23-2021-08-notional (1Positive-SP) (1Negative-FLP)** with the repository **https://github.com/code-423n4/2021-08-notional** on branch **default branch**. The assessment was carried out by scanning the project's codebase using the scan engine **AI Analyzer**. There are in total **4** vulnerabilities / security risks discovered during the scanning session, among which **0** critical vulnerabilities, **4** high risk vulnerabilities, **0** medium risk vulnerabilities, **0** low risk vulnerabilities, **0** informational issues.

ID	Description	Severity
MSA-001	MWE-206: No Slippage Limit Check	High risk
MSA-002	MWE-206: No Slippage Limit Check	High risk
MSA-003	MWE-206: No Slippage Limit Check	High risk
MSA-004	MWE-206: No Slippage Limit Check	High risk

Findings

Critical (0)

No Critical vulnerabilities found here

High risk (4)

1. MWE-206: No Slippage Limit Check

High risk Security Analyzer

No slippage limit check was performed to prevent sandwich attacks.

File(s) Affected

contracts/mocks/MockMarket.sol #127-141

```
    function removeLiquidity(MarketParameters memory marketState, int256 tokensToRemove)
        public
        pure
        returns (
            MarketParameters memory,
            int256,
            int256
        )
    {
        (int256 assetCash, int256 fCash) = marketState.removeLiquidity(tokensToRemove);

        assert(assetCash >= 0);
        assert(fCash >= 0);
        return (marketState, assetCash, fCash);
    }
```

contracts/internal/liquidation/LiquidateCurrency.sol #440-502

```
    function _withdrawCollateralLiquidityTokens(
        PortfolioState memory portfolioState,
        LiquidationFactors memory factors,
        uint256 blockTime,
        int256 collateralToWithdraw
    ) internal view returns (int256) {
        require(portfolioState.newAssets.length == 0); // dev: new assets in portfolio
        factors.markets = new MarketParameters[](factors.cashGroup.maxMarketIndex);

        for (uint256 i; i < portfolioState.storedAssets.length; i++) {
            PortfolioAsset memory asset = portfolioState.storedAssets[i];
            if (asset.storageState == AssetStorageState.Delete) continue;
            if (
                !AssetHandler.isLiquidityToken(asset.assetType) ||
                asset.currencyId != factors.cashGroup.currencyId
            ) continue;

            uint256 marketIndex = asset.assetType - 1;
            // This is set up this way so that we can delay setting storage of markets so that this met
            // remain a view function
            factors.cashGroup.loadMarket(
                factors.markets[marketIndex - 1],
                marketIndex,
                true,
                blockTime
            );
            (int256 cashClaim, int256 fCashClaim) =
                asset.getCashClaims(factors.markets[marketIndex - 1]);

            if (cashClaim <= collateralToWithdraw) {
                // The additional cash is insufficient to cover asset amount required so we just remove
                portfolioState.deleteAsset(i);
                factors.markets[marketIndex - 1].removeLiquidity(asset.notional);

                // overflow checked above
                collateralToWithdraw = collateralToWithdraw - cashClaim;
            } else {
                // Otherwise remove a proportional amount of liquidity tokens to cover the amount remai
                // NOTE: dust can accrue when withdrawing liquidity at this point
                int256 tokensToRemove = asset.notional.mul(collateralToWithdraw).div(cashClaim);
                (cashClaim, fCashClaim) = factors.markets[marketIndex - 1].removeLiquidity(
                    tokensToRemove
                );

                // Remove liquidity token balance
                portfolioState.storedAssets[i].notional = asset.notional.subNoNeg(tokensToRemove);
                portfolioState.storedAssets[i].storageState = AssetStorageState.Update;
                collateralToWithdraw = 0;
            }

            // Add the netfCash asset to the portfolio since we've withdrawn the liquidity tokens
            portfolioState.addAsset(
                factors.cashGroup.currencyId,
                asset.maturity,
                Constants.FCASH_ASSET_TYPE,
                fCashClaim
            );

            if (collateralToWithdraw == 0) return 0;
        }

        return collateralToWithdraw;
    }
```

Recommendation

Add slippage limit check when do liquidity-related operations.

2. MWE-206: No Slippage Limit Check

High risk

Security Analyzer

No slippage limit check was performed to prevent sandwich attacks.

File(s) Affected

contracts/internal/liquidation/LiquidateCurrency.sol #334-416

```
    function _withdrawLocalLiquidityTokens(
        PortfolioState memory portfolioState,
        LiquidationFactors memory factors,
        uint256 blockTime,
        int256 assetAmountRemaining
    ) internal view returns (WithdrawFactors memory, int256) {
        require(portfolioState.newAssets.length == 0); // dev: new assets in portfolio
        factors.markets = new MarketParameters[](factors.cashGroup.maxMarketIndex);
        // Do this to deal with stack issues
        WithdrawFactors memory w;

        for (uint256 i; i < portfolioState.storedAssets.length; i++) {
            PortfolioAsset memory asset = portfolioState.storedAssets[i];
            if (asset.storageState == AssetStorageState.Delete) continue;
            if (
                !AssetHandler.isLiquidityToken(asset.assetType) ||
                asset.currencyId != factors.cashGroup.currencyId
            ) continue;

            uint256 marketIndex = asset.assetType - 1;
            // This is set up this way so that we can delay setting storage of markets so that this met
            // remain a view function
            factors.cashGroup.loadMarket(
                factors.markets[marketIndex - 1],
                marketIndex,
                true,
                blockTime
            );

            // NOTE: we do not give any credit to the haircut fCash in this procedure but it will end u
            // additional collateral value back into the account. It's probably too complex to deal wit
            // we will just leave it as such.
            (w.assetCash, w.fCash) = asset.getCashClaims(factors.markets[marketIndex - 1]);
            _calculateNetCashIncreaseAndIncentivePaid(factors, w, asset.assetType);

            // (netCashToAccount <= assetAmountRemaining)
            if (w.netCashIncrease.subNoNeg(w.incentivePaid) <= assetAmountRemaining) {
                // The additional cash is insufficient to cover asset amount required so we just remove
                portfolioState.deleteAsset(i);
                factors.markets[marketIndex - 1].removeLiquidity(asset.notional);

                // assetAmountRemaining = assetAmountRemaining - netCashToAccount
                // netCashToAccount = netCashIncrease - incentivePaid
                // overflow checked above
                assetAmountRemaining =
                    assetAmountRemaining -
                    w.netCashIncrease.sub(w.incentivePaid);
            } else {
                // Otherwise remove a proportional amount of liquidity tokens to cover the amount remai
                int256 tokensToRemove =
                    asset.notional.mul(assetAmountRemaining).div(
                        w.netCashIncrease.subNoNeg(w.incentivePaid)
                    );

                (w.assetCash, w.fCash) = factors.markets[marketIndex - 1].removeLiquidity(
                    tokensToRemove
                );
                // Recalculate net cash increase and incentive paid. w.assetCash is different because
                // remove asset cash
                _calculateNetCashIncreaseAndIncentivePaid(factors, w, asset.assetType);

                // Remove liquidity token balance
                portfolioState.storedAssets[i].notional = asset.notional.subNoNeg(tokensToRemove);
                portfolioState.storedAssets[i].storageState = AssetStorageState.Update;
                assetAmountRemaining = 0;
            }

            w.totalIncentivePaid = w.totalIncentivePaid.add(w.incentivePaid);
            w.totalCashClaim = w.totalCashClaim.add(w.assetCash);

            // Add the netfCash asset to the portfolio since we've withdrawn the liquidity tokens
            portfolioState.addAsset(
                factors.cashGroup.currencyId,
                asset.maturity,
                Constants.FCASH_ASSET_TYPE,
                w.fCash
            );

            if (assetAmountRemaining == 0) break;
        }

        return (w, assetAmountRemaining);
    }
```

Recommendation

Add slippage limit check when do liquidity-related operations.

3. MWE-206: No Slippage Limit Check

High risk Security Analyzer

No slippage limit check was performed to prevent sandwich attacks.

File(s) Affected

contracts/internal/liquidation/LiquidateCurrency.sol #334-416

```
    function _withdrawLocalLiquidityTokens(
        PortfolioState memory portfolioState,
        LiquidationFactors memory factors,
        uint256 blockTime,
        int256 assetAmountRemaining
    ) internal view returns (WithdrawFactors memory, int256) {
        require(portfolioState.newAssets.length == 0); // dev: new assets in portfolio
        factors.markets = new MarketParameters[](factors.cashGroup.maxMarketIndex);
        // Do this to deal with stack issues
        WithdrawFactors memory w;

        for (uint256 i; i < portfolioState.storedAssets.length; i++) {
            PortfolioAsset memory asset = portfolioState.storedAssets[i];
            if (asset.storageState == AssetStorageState.Delete) continue;
            if (
                !AssetHandler.isLiquidityToken(asset.assetType) ||
                asset.currencyId != factors.cashGroup.currencyId
            ) continue;

            uint256 marketIndex = asset.assetType - 1;
            // This is set up this way so that we can delay setting storage of markets so that this met
            // remain a view function
            factors.cashGroup.loadMarket(
                factors.markets[marketIndex - 1],
                marketIndex,
                true,
                blockTime
            );

            // NOTE: we do not give any credit to the haircut fCash in this procedure but it will end u
            // additional collateral value back into the account. It's probably too complex to deal wit
            // we will just leave it as such.
            (w.assetCash, w.fCash) = asset.getCashClaims(factors.markets[marketIndex - 1]);
            _calculateNetCashIncreaseAndIncentivePaid(factors, w, asset.assetType);

            // (netCashToAccount <= assetAmountRemaining)
            if (w.netCashIncrease.subNoNeg(w.incentivePaid) <= assetAmountRemaining) {
                // The additional cash is insufficient to cover asset amount required so we just remove
                portfolioState.deleteAsset(i);
                factors.markets[marketIndex - 1].removeLiquidity(asset.notional);

                // assetAmountRemaining = assetAmountRemaining - netCashToAccount
                // netCashToAccount = netCashIncrease - incentivePaid
                // overflow checked above
                assetAmountRemaining =
                    assetAmountRemaining -
                    w.netCashIncrease.sub(w.incentivePaid);
            } else {
                // Otherwise remove a proportional amount of liquidity tokens to cover the amount remai
                int256 tokensToRemove =
                    asset.notional.mul(assetAmountRemaining).div(
                        w.netCashIncrease.subNoNeg(w.incentivePaid)
                    );

                (w.assetCash, w.fCash) = factors.markets[marketIndex - 1].removeLiquidity(
                    tokensToRemove
                );
                // Recalculate net cash increase and incentive paid. w.assetCash is different because
                // remove asset cash
                _calculateNetCashIncreaseAndIncentivePaid(factors, w, asset.assetType);

                // Remove liquidity token balance
                portfolioState.storedAssets[i].notional = asset.notional.subNoNeg(tokensToRemove);
                portfolioState.storedAssets[i].storageState = AssetStorageState.Update;
                assetAmountRemaining = 0;
            }

            w.totalIncentivePaid = w.totalIncentivePaid.add(w.incentivePaid);
            w.totalCashClaim = w.totalCashClaim.add(w.assetCash);

            // Add the netfCash asset to the portfolio since we've withdrawn the liquidity tokens
            portfolioState.addAsset(
                factors.cashGroup.currencyId,
                asset.maturity,
                Constants.FCASH_ASSET_TYPE,
                w.fCash
            );

            if (assetAmountRemaining == 0) break;
        }

        return (w, assetAmountRemaining);
    }
```

contracts/internal/liquidation/LiquidateCurrency.sol #50-130

```
    function liquidateLocalCurrency(
        uint256 localCurrency,
        uint96 maxNTokenLiquidation,
        uint256 blockTime,
        BalanceState memory balanceState,
        LiquidationFactors memory factors,
        PortfolioState memory portfolio
    ) internal view returns (int256) {
        require(factors.localAssetAvailable < 0, "No local debt");

        int256 assetBenefitRequired =
            factors.cashGroup.assetRate.convertFromUnderlying(
                factors
                    .localETHRate
                    .convertETHTo(factors.netETHValue.neg())
                    .mul(Constants.PERCENTAGE_DECIMALS)
                    .div(factors.localETHRate.buffer)
            );

        int256 netAssetCashFromLiquidator;

        if (_hasLiquidityTokens(portfolio.storedAssets, localCurrency)) {
            WithdrawFactors memory w;
            (w, assetBenefitRequired) = _withdrawLocalLiquidityTokens(
                portfolio,
                factors,
                blockTime,
                assetBenefitRequired
            );
            netAssetCashFromLiquidator = w.totalIncentivePaid.neg();
            balanceState.netCashChange = w.totalCashClaim.sub(w.totalIncentivePaid);
        }

        if (factors.nTokenHaircutAssetValue > 0) {
            int256 nTokensToLiquidate;
            {
                // This will not underflow, checked when saving parameters
                int256 haircutDiff =
                    int256(
                        uint8(factors.nTokenParameters[Constants.LIQUIDATION_HAIRCUT_PERCENTAGE]) -
                            uint8(factors.nTokenParameters[Constants.PV_HAIRCUT_PERCENTAGE])
                    ) * Constants.PERCENTAGE_DECIMALS;

                // fullNTokenPV = haircutTokenPV / haircutPercentage
                // benefitGained = nTokensToLiquidate * (liquidatedPV - freeCollateralPV)
                // benefitGained = nTokensToLiquidate * (fullNTokenPV * liquidatedPV - fullNTokenPV * pv
                // benefitGained = nTokensToLiquidate * fullNTokenPV * (liquidatedPV - pvHaircut) / tota
                // benefitGained = nTokensToLiquidate * (haircutTokenPV / haircutPercentage) * (liquidat
                // benefitGained = nTokensToLiquidate * haircutTokenPV * (liquidationHaircut - pvHaircut
                // nTokensToLiquidate = (benefitGained * totalBalance * haircutPercentage) / (haircutTok
                 nTokensToLiquidate = assetBenefitRequired
                     .mul(balanceState.storedNTokenBalance)
                     .mul(int256(uint8(factors.nTokenParameters[Constants.PV_HAIRCUT_PERCENTAGE])))
                     .div(factors.nTokenHaircutAssetValue.mul(haircutDiff));
             }

             nTokensToLiquidate = LiquidationHelpers.calculateLiquidationAmount(
                nTokensToLiquidate,
                balanceState.storedNTokenBalance,
                int256(maxNTokenLiquidation)
            );
            balanceState.netNTokenTransfer = nTokensToLiquidate.neg();

            {
                // fullNTokenPV = haircutTokenPV / haircutPercentage
                // localFromLiquidator = tokensToLiquidate * fullNTokenPV * liquidationHaircut / totalB
                // prettier-ignore
                int256 localAssetCash =
                    nTokensToLiquidate
                        .mul(int256(uint8(factors.nTokenParameters[Constants.LIQUIDATION_HAIRCUT_PERCEN
                        .mul(factors.nTokenHaircutAssetValue)
                        .div(int256(uint8(factors.nTokenParameters[Constants.PV_HAIRCUT_PERCENTAGE])))
                        .div(balanceState.storedNTokenBalance);

                balanceState.netCashChange = balanceState.netCashChange.add(localAssetCash);
                netAssetCashFromLiquidator = netAssetCashFromLiquidator.add(localAssetCash);
            }
        }

        return netAssetCashFromLiquidator;
    }
```

Recommendation

Add slippage limit check when do liquidity-related operations.

4. MWE-206: No Slippage Limit Check

High risk Security Analyzer

No slippage limit check was performed to prevent sandwich attacks.

File(s) Affected

contracts/internal/liquidation/LiquidateCurrency.sol #440-502

```
    function _withdrawCollateralLiquidityTokens(
        PortfolioState memory portfolioState,
        LiquidationFactors memory factors,
        uint256 blockTime,
        int256 collateralToWithdraw
    ) internal view returns (int256) {
        require(portfolioState.newAssets.length == 0); // dev: new assets in portfolio
        factors.markets = new MarketParameters[](factors.cashGroup.maxMarketIndex);

        for (uint256 i; i < portfolioState.storedAssets.length; i++) {
            PortfolioAsset memory asset = portfolioState.storedAssets[i];
            if (asset.storageState == AssetStorageState.Delete) continue;
            if (
                !AssetHandler.isLiquidityToken(asset.assetType) ||
                asset.currencyId != factors.cashGroup.currencyId
            ) continue;

            uint256 marketIndex = asset.assetType - 1;
            // This is set up this way so that we can delay setting storage of markets so that this met
            // remain a view function
            factors.cashGroup.loadMarket(
                factors.markets[marketIndex - 1],
                marketIndex,
                true,
                blockTime
            );
            (int256 cashClaim, int256 fCashClaim) =
                asset.getCashClaims(factors.markets[marketIndex - 1]);

            if (cashClaim <= collateralToWithdraw) {
                // The additional cash is insufficient to cover asset amount required so we just remove
                portfolioState.deleteAsset(i);
                factors.markets[marketIndex - 1].removeLiquidity(asset.notional);

                // overflow checked above
                collateralToWithdraw = collateralToWithdraw - cashClaim;
            } else {
                // Otherwise remove a proportional amount of liquidity tokens to cover the amount remai
                // NOTE: dust can accrue when withdrawing liquidity at this point
                int256 tokensToRemove = asset.notional.mul(collateralToWithdraw).div(cashClaim);
                (cashClaim, fCashClaim) = factors.markets[marketIndex - 1].removeLiquidity(
                    tokensToRemove
                );

                // Remove liquidity token balance
                portfolioState.storedAssets[i].notional = asset.notional.subNoNeg(tokensToRemove);
                portfolioState.storedAssets[i].storageState = AssetStorageState.Update;
                collateralToWithdraw = 0;
            }

            // Add the netfCash asset to the portfolio since we've withdrawn the liquidity tokens
            portfolioState.addAsset(
                factors.cashGroup.currencyId,
                asset.maturity,
                Constants.FCASH_ASSET_TYPE,
                fCashClaim
            );

            if (collateralToWithdraw == 0) return 0;
        }

        return collateralToWithdraw;
    }
```

contracts/internal/liquidation/LiquidateCurrency.sol #134-213

```
    function liquidateCollateralCurrency(
        uint128 maxCollateralLiquidation,
        uint96 maxNTokenLiquidation,
        uint256 blockTime,
        BalanceState memory balanceState,
        LiquidationFactors memory factors,
        PortfolioState memory portfolio
    ) internal view returns (int256) {
        require(factors.localAssetAvailable < 0, "No local debt");
        require(factors.collateralAssetAvailable > 0, "No collateral");

        (
            int256 requiredCollateralAssetCash,
            int256 localAssetCashFromLiquidator,
            int256 liquidationDiscount
        ) = _calculateCollateralToRaise(factors, int256(maxCollateralLiquidation));

        int256 collateralAssetRemaining = requiredCollateralAssetCash;
        if (balanceState.storedCashBalance > 0) {
            if (balanceState.storedCashBalance > collateralAssetRemaining) {
                balanceState.netCashChange = collateralAssetRemaining.neg();
                collateralAssetRemaining = 0;
            } else {
                // Sell off all cash balance and calculate remaining collateral
                balanceState.netCashChange = balanceState.storedCashBalance.neg();
                collateralAssetRemaining = collateralAssetRemaining.sub(
                    balanceState.storedCashBalance
                );
            }
        }

        if (
            collateralAssetRemaining > 0 &&
            _hasLiquidityTokens(portfolio.storedAssets, balanceState.currencyId)
        ) {
            int256 newCollateralAssetRemaining =
                _withdrawCollateralLiquidityTokens(
                    portfolio,
                    factors,
                    blockTime,
                    collateralAssetRemaining
                );

            // This is a hack and ugly but there are stack issues in `LiquidateCurrencyAction.liquidate
            // and this is a way to deal with it with the fewest contortions. There are no asset cash t
            // so we overload the meaning of the field here to hold the net liquidity token cash change
            // going into finalize for the liquidated account's cash balances. This value is not simply
            // because the cashClaim value is not stored in the balances and therefore the liquidated a
            // debited from their stored cash value.
            balanceState.netAssetTransferInternalPrecision = collateralAssetRemaining.sub(
                newCollateralAssetRemaining
            );
            collateralAssetRemaining = newCollateralAssetRemaining;
        }

        if (collateralAssetRemaining > 0 && factors.nTokenHaircutAssetValue > 0) {
            collateralAssetRemaining = _calculateCollateralNTokenTransfer(
                balanceState,
                factors,
                collateralAssetRemaining,
                int256(maxNTokenLiquidation)
            );
        }

        if (collateralAssetRemaining > 0) {
            // If there is any collateral asset remaining then recalculate the localAssetCashFromLiquic
            // prettier-ignore
            (
                /* collateralToRaise */,
                localAssetCashFromLiquidator
            ) = LiquidationHelpers.calculateLocalToPurchase(
                factors,
                liquidationDiscount,
                requiredCollateralAssetCash.sub(collateralAssetRemaining),
                requiredCollateralAssetCash.sub(collateralAssetRemaining)
            );
        }

        return localAssetCashFromLiquidator;
    }
```

Recommendation

Add slippage limit check when do liquidity-related operations.

Medium risk (0)

No Medium risk vulnerabilities found here

Low risk (0)

No Low risk vulnerabilities found here

Informational (0)

No Informational vulnerabilities found here

Disclaimer

MetaTrust HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, MetaTrust SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, MetaTrust MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, MetaTrust PROVIDES NO WARRANTY OR UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER MetaTrust NOR ANY OF MetaTrust'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. MetaTrust WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT MetaTrust'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING 23-2021-08-notional (1Positive-SP) (1Negative-FLP) Security Assessment MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST MetaTrust WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF MetaTrust CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST MetaTrust WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.