HASH SECURITY

Copra Finance Security Review

Auditors

hash

1 Executive Summary

Over the course of 1 days in total, Copra Finance engaged with hash to review Copra Finance-KTX Integration. In this period of time a total of 12 issues were found.

Summary

Project Name	Copra Finance
Code Under Review	711825e74c...
Type of Project	Lending, DeFi
Audit Timeline	1 June 2024 - 2 June 2024

Issues Found

High Risk	1
Medium Risk	5
Low Risk	2
Informational	4
Total Issues	12

2 Findings

2.1 High Risk

2.1.1 Incorrect decimal scaling in `_getDeployedAssetValue`

Description: In `_getDeployedAssetValue`, the final value is returned as `navUSD18DP * WAD / baseAssetOraclePrice` where `baseAssetOraclePrice` is in 18 decimals and `WAD` is set to 1e18

```
    function _getDeployedAssetValue(address rewardRouterAddr) internal view override returns (uint256) {

        .....

        uint256 baseAssetOraclePrice = _getAssetOraclePrice(address(s_terms.asset));
        return navUSD18DP * WAD / baseAssetOraclePrice;
    }
```

This is incorrect as it should've been converted to the asset's decimal which can be vary from 18 (eg: WBTC is 8)

POC

Apply the following diff and run `testHashIncorrectScaledNetAssetValue`

```
diff --git a/test/integration/KTXIntegrationTest.t.sol b/test/integration/KTXIntegrationTest.t.sol
index e244911..3095d80 100644
--- a/test/integration/KTXIntegrationTest.t.sol
+++ b/test/integration/KTXIntegrationTest.t.sol
@@ -1,7 +1,7 @@
 // SPDX-License-Identifier: UNLICENSED
 pragma solidity 0.8.19;

-import {Test} from "forge-std/Test.sol";
+import {Test,console2} from "forge-std/Test.sol";
 import {IntegrationTestConstants} from "./IntegrationTestConstants.t.sol";
 import {IERC20} from "openzeppelin/token/ERC20/IERC20.sol";
 import {LiquidityWarehouse} from "../../src/LiquidityWarehouse.sol";
@@ -766,6 +766,7 @@ contract KTXIntegrationTest_WhenWETHLiquidityDeployed is KTXIntegrationTest {
 }

 contract KTXIntegrationTest_WhenWBTCLiquidityDeployed is KTXIntegrationTest {
+    uint beforeDeployBalance;
     function setUp() public override {
         KTXIntegrationTest.setUp();

@@ -798,6 +799,8 @@ contract KTXIntegrationTest_WhenWBTCLiquidityDeployed is KTXIntegrationTest {
             value: 0
         });

+        beforeDeployBalance = s_wbtc.balanceOf(address(s_wbtcKTXLiquidityWarehouse));
+
         deployActions[1] = LiquidityWarehouseAccessControl.ExecuteAction({
             target: REWARD_ROUTER,
             data: abi.encodeWithSelector(
@@ -856,6 +859,15 @@ contract KTXIntegrationTest_WhenWBTCLiquidityDeployed is KTXIntegrationTest {
         vm.warp(block.timestamp + 300);
     }

+    function testHashIncorrectScaledNetAssetValue() public {
+        uint netAssetValueIncorrect = s_wbtcKTXLiquidityWarehouse.getNetAssetValue();
+        console2.log("beforeDeployBalance",beforeDeployBalance);
+        console2.log("netAssetValueIncorrect",netAssetValueIncorrect);
+
+        assert(beforeDeployBalance == 60000000);
+        assert(netAssetValueIncorrect == 598704367517294257);
+    }
+
     function test_CanWithdrawWBTC() public {
         uint256 balanceBefore = s_wbtc.balanceOf(LENDER_ONE);
         vm.startPrank(LENDER_ONE);
```

Recommendation: Convert to asset's decimal instead of 18

2.2 Medium Risk

2.2.1 getSwapFeeBasisPoints is used for fee calculation instead of getSellUsdgFeeBasisPoints

To redemption fees is calculated inside the warehouse by invoking `getSwapFeeBasisPoints` on the vault utils contract

link

```
    function _counterKtxFee(address vault, address asset, uint256 klpAmount, uint256 usdgAmount)
        internal
        view
        returns (uint256)
    {
        return (
            klpAmount
                * (
                    KTX_FEE_PRECISION * KLP_PRICE_PRECISION
                        / (
                            KTX_FEE_PRECISION
                                - (
=>                                  IVaultUtils(IVault(vault).vaultUtils()).getSwapFeeBasisPoints(
                                        IVault(vault).usdg(), asset, usdgAmount
                                    )
                                )
                        )
                ) / KLP_PRICE_PRECISION
        ) - klpAmount;
    }
```

This is inconsistent from how the actual fees will be calculated at the time of liquidity removal which uses `getSellUsdgFeeBasisPoints` instead to compute the fees

```
        uint256 feeBasisPoints = vaultUtils.getSellUsdgFeeBasisPoints(
            _token,
            usdgAmount
        );
```

POC:

```
function testSwapFeeVsSellFee() public {
  vm.createSelectFork(vm.envString("ARB_RPC"));
  vm.rollFork(217107023);

    assert(address(c) == 0xc657A1440d266dD21ec3c299A8B9098065f663Bb);

    IVaultUtils vaultUtils = IVaultUtils(0xbde9c699e719bb44811252FDb3B37E6D3eDa5a28);

    uint256 sellFeeBasisPoints = vaultUtils.getSellUsdgFeeBasisPoints(
          WETH,
          4000e18
      );
      uint256 swapFeeBasisPoints = vaultUtils.getSwapFeeBasisPoints(c.usdg(),
          WETH,
          4000e18
      );

    //console.log("sellFeeBasisPoints",sellFeeBasisPoints);
    //console.log("swapFeeBasisPoints",swapFeeBasisPoints);
    assert(sellFeeBasisPoints == 16);
    assert(swapFeeBasisPoints == 40);
}
```

Recommendation: Use `getSellUsdgFeeBasisPoints` to compute the fees instead

2.2.2 Withdrawals close to full amount cannot succeed due to early return and higher valuation of the asset

To calculate the amount of klp tokens to burn for `withdrawAmount` of assets, the max price of the asset is taken. This can cause the `klpToBurn` amount to be greater than the staked balance (using which the calculation chain had begun)

```
    function _withdrawFromTarget(uint256 withdrawAmount, address rewardRouterAddr, bytes memory)
        internal
        override
        returns (bool)
    {

        .....

        if (withdrawAmount == type(uint256).max) {
            klpToBurn =
↪   IRewardTracker(IRewardRouter(rewardRouterAddr).feeKlpTracker()).stakedAmounts(address(this));
        } else {
=>          klpToBurn =
                _convertToUSDG(ktxVault, address(s_terms.asset), withdrawAmount) * KLP_PRICE_PRECISION
↪   / klpPrice;

=>          if (
                klpToBurn >
↪   IRewardTracker(IRewardRouter(rewardRouterAddr).feeKlpTracker()).stakedAmounts(address(this))
            ) {
                return false;
```

getMaxPrice is used for valuing the asset here

```
    function _convertToUSDG(address vault, address asset, uint256 amount) internal view returns
↪   (uint256) {
=>      uint256 ktxUsdRate = IVault(vault).getMaxPrice(asset);
        return amount * ktxUsdRate / 10 ** (KLP_DECIMAL - (USDG_DECIMAL -
↪   IERC20Metadata(asset).decimals()));
    }
```

In case of such scenario's, the contract returns ealy by returning `false` without attempting to redeem any tokena and hence any sensible value kept of maxLoss amount will cause the tx to revert For this, the `getMaxPrice` should return a higher price than the `getMinPrice` function which is not the case currently as the feed is configured with `0` for the spreadPoints. But this value is settable and might change in future

POC

```
function testHash_FullWithdrawalsCanRevertDueToHigherThanStakingAmount() public {
        // ktx price feed gov: 0x32f4a5BbFA839cDEeCa2f86433C6825CB4Fbb3FD
        // @audit there is some other mistake due to which it is even reverting without setting the fee
↪   spread points
        address VAULT_PRICE_FEED = 0x28403B8668Db61De7484A2EAafB65b950a21a2fb;
        address PRICE_FEED_GOV = 0x32f4a5BbFA839cDEeCa2f86433C6825CB4Fbb3FD;

        vm.startPrank(PRICE_FEED_GOV);

        // setting 50, currently this is 0
        IVaultPriceFeed(VAULT_PRICE_FEED).setSpreadBasisPoints(WETH,50);
        vm.stopPrank();

        uint256  LENDER_LP_TOKEN_ID = 0;
        uint256  BORROWER_LP_TOKEN_ID = 1;

        vm.startPrank(LENDER_ONE);

        uint lenderShares = s_wethKTXLiquidityWarehouse.balanceOf(LENDER_ONE,LENDER_LP_TOKEN_ID);
        s_wethKTXLiquidityWarehouse.withdrawLender(
            lenderShares, abi.encode(REWARD_ROUTER)
        );

        vm.stopPrank();

        vm.startPrank(FEE_RECIPIENT);

        s_wethKTXLiquidityWarehouse.withdrawLenderFees(
            abi.encode(REWARD_ROUTER)        );

        vm.stopPrank();

        vm.startPrank(BORROWER_ONE);

        uint borrowerShares = s_wethKTXLiquidityWarehouse.balanceOf(BORROWER_ONE,BORROWER_LP_TOKEN_ID);

        vm.expectRevert(); // InsufficientLiquidity. No withdrawal attempt will be made due to the
↪   early return
        s_wethKTXLiquidityWarehouse.withdrawBorrower(
            borrowerShares, abi.encode(0.1e18) // 0.5 eth as max loss with deposit of 1 eth
        );
    }
```

The klpToBurn calculated here is 2360321038283876268567 while the stakedBalance is only

23485781475461455411 92

2.2.3 Lack of proper checks to determine remaining burnable KLP amount can cause reverts

When withdrawing from KTX, the checks kept to determine how much further klp can be burned is not sufficient

link

```
    function _burnKLP(
        address asset,
        uint256 klpBurnAmount,
        uint256 klpPrice,
        uint256 klpPricePrecision,
        address ktxVault,
        IRewardRouter rewardRouter
    ) internal returns (uint256) {

        .....

        /// return amount of remaining klp to burn
        if (counterKTXVaultFee == 0 &&
 ↪  IRewardTracker(rewardRouter.feeKlpTracker()).stakedAmounts(address(this)) != 0) {
            return klpBurnAmount - klpAmount + klpRedeemFee + counterSwapFee;
        }
```

In case the addition of (redemptionFee + swapFee) exceeded the staked balance of the warehouse earlier, this amount is not attempted to be withdrawn from KTX. But if after the withdrawal, the staked balance is greater than zero but not enough to cover the (redemptionFee + swapFee), it is still attempted to be withdrawn from ktx causing a revert

POC

Add the following test and run `forge test --mt testHash_FullWithdrawalsCanRevertDueToFeesCheck`

```
    function testHash_FullWithdrawalsCanRevertDueToFeesCheck() public {

        uint256  LENDER_LP_TOKEN_ID = 0;
        uint256  BORROWER_LP_TOKEN_ID = 1;

        vm.startPrank(LENDER_ONE);

        uint lenderShares = s_wethKTXLiquidityWarehouse.balanceOf(LENDER_ONE,LENDER_LP_TOKEN_ID);
        s_wethKTXLiquidityWarehouse.withdrawLender(
            lenderShares, abi.encode(REWARD_ROUTER)
        );

        vm.stopPrank();

        vm.startPrank(FEE_RECIPIENT);

        s_wethKTXLiquidityWarehouse.withdrawLenderFees(
            abi.encode(REWARD_ROUTER)        );

        vm.stopPrank();

        vm.startPrank(BORROWER_ONE);

        uint borrowerShares = s_wethKTXLiquidityWarehouse.balanceOf(BORROWER_ONE,BORROWER_LP_TOKEN_ID);

        vm.expectRevert("RewardTracker: _amount exceeds stakedAmount");
        s_wethKTXLiquidityWarehouse.withdrawBorrower(
            borrowerShares, abi.encode(REWARD_ROUTER)
        );
    }
```

Recommendation Rewrite the logic for comparison with the stakedAmount and the feeAmount

2.2.4 Reserved amount is not considered when computing the withdrawable amount

Description: Currently the `maxBurnableKLPAmount` is calculated considering the entire token balance of ktxVault to be withdrawable

link

```
    function _burnKLP(
        address asset,
        uint256 klpBurnAmount,
        uint256 klpPrice,
        uint256 klpPricePrecision,
        address ktxVault,
        IRewardRouter rewardRouter
    ) internal returns (uint256) {
        uint256 ktxVaultBalance = IERC20Metadata(asset).balanceOf(ktxVault);
        uint256 maxBurnableKLPAmount = _convertToUSDG(ktxVault, asset, ktxVaultBalance) *
↪   klpPricePrecision / klpPrice;
```

But this is incorrect as only `poolAmounts[_token] - reservedAmounts[_token]` is allowed to be withdrawn. Attempting to withdraw more will result in a revert

```
  function _decreasePoolAmount(address _token, uint256 _amount) private {
      poolAmounts[_token] = poolAmounts[_token].sub(
          _amount,
          "Vault: poolAmount exceeded"
      );
      _validate(reservedAmounts[_token] <= poolAmounts[_token], 50);
      emit DecreasePoolAmount(_token, _amount);
  }
```

POC:

```
function testHashWithdrawableAmountExceedsReservePOC() public {
    IRewardRouter c = IRewardRouter(0x9d4c701A5D06a1cd73853ebd3F8d4b32C153bD9A);

    vm.createSelectFork(vm.envString("ARB_RPC"));
    vm.rollFork(217107023);

    // token balance is high but withdrawable amount is low (pooledAmount - reservedAmount)
    IVault vault = IVault(0xc657A1440d266dD21ec3c299A8B9098065f663Bb);

    uint whitelistedTokenCount = vault.allWhitelistedTokensLength();

    address maxDiffToken;
    uint maxDiff;
    for(uint i=0;i<whitelistedTokenCount;i++){
      address token = vault.allWhitelistedTokens(i);
      uint tokenBalance = IERC20(token).balanceOf(address(vault));
      uint actualWithdrawable = vault.poolAmounts(token) - vault.reservedAmounts(token);
      if(tokenBalance - actualWithdrawable > maxDiff){
        maxDiff = tokenBalance - actualWithdrawable;
        maxDiffToken = token;
      }
    }

    console.log("max diff token",maxDiffToken);
    console.log("max diff",maxDiff);

    uint tokenBalance = IERC20(maxDiffToken).balanceOf(address(vault));

    IKlpManager klpManager = IKlpManager(0xfF0255c564F810a5108F00be199600cF520507Ca);

    uint256 klpPrice = klpManager.getPrice(false);
    uint256 KLP_PRICE_PRECISION = 1e30;

    uint256 calculatedMaxBurnableKLPAmount = _convertToUSDG(address(vault), maxDiffToken,
↪  tokenBalance) * KLP_PRICE_PRECISION / klpPrice;

    uint actualWithdrawableTokenBalance = vault.poolAmounts(maxDiffToken) -
↪  vault.reservedAmounts(maxDiffToken);
    uint actualWithdrawableKLPAmount = _convertToUSDG(address(vault), maxDiffToken,
↪  actualWithdrawableTokenBalance) * KLP_PRICE_PRECISION / klpPrice;

    //uint actualWithdrawable ==
    // klp whale https://arbiscan.io/address/0x57475770bc41c817ea607c2ed757ea006db4adda
    address klpWhale = 0x57475770BC41C817eA607C2Ed757EA006dB4AdDA;

    console.log("tokenBalance",tokenBalance);
    console.log("actualWithdrawableTokenBalance",actualWithdrawableTokenBalance);
    console.log("pool amount",vault.poolAmounts(maxDiffToken));
```

```
        console.log("calculatedMaxBurnableKLPAmount",calculatedMaxBurnableKLPAmount);
        console.log("actualWithdrawableKLPAmount",actualWithdrawableKLPAmount);

        console.log("balance of whale",IERC20(c.stakedKlpTracker()).balanceOf(klpWhale));
        console.log("totalSupply",IERC20(c.stakedKlpTracker()).totalSupply());

        assert(IERC20(c.stakedKlpTracker()).balanceOf(klpWhale) > calculatedMaxBurnableKLPAmount);

        // trying to withdraw calculatedMaxBurnableKLPAmount will revert
        vm.startPrank(klpWhale);

        // - 500e18, else the pool amount will itself get exceeded
        vm.expectRevert("Vault: reserve exceeds pool");
        c.unstakeAndRedeemKlp(maxDiffToken,calculatedMaxBurnableKLPAmount - 500e18,0,klpWhale);
    }
```

Recommendation: Use `poolAmounts[_token] - reservedAmounts[_token]` instead of taking the entire token balance

2.2.5 Using allWhitelistedTokens to fetch the list of tokens can cause reverts in future

Description: To fetch the list of assets inside the vault for withdrawing KLP, the `allWhitelistedTokens` array is used. But this can be flawed as the dewhitelisted tokens are not updated in the `allWhitelistedTokens` array

```
        uint256 numWhitelistedTokens = IVault(ktxVault).allWhitelistedTokensLength();


        /// Withdraw from KTX vault until withdraw amount is fulfilled
        for (uint256 i; i < numWhitelistedTokens; ++i) {
            address asset = IVault(ktxVault).allWhitelistedTokens(i);
```

function to dewhitelist a token inside the vault

```
    function clearTokenConfig(address _token) external {
        _onlyGov();
        _validate(whitelistedTokens[_token], 13);
        totalTokenWeights = totalTokenWeights.sub(tokenWeights[_token]);
        delete whitelistedTokens[_token];
        delete tokenDecimals[_token];
        delete tokenWeights[_token];
        delete minProfitBasisPoints[_token];
        delete maxUsdgAmounts[_token];
        delete stableTokens[_token];
        delete shortableTokens[_token];
        whitelistedTokenCount = whitelistedTokenCount.sub(1);
    }
```

Recommendation: Confirm if the token is still whitelisted before attempting to withdraw

2.3 Low

2.3.1 Redeem fee and swap fee estimation is incorrect

Description: Once the klpAmount has been calculated the `redeem` and `swap` fee estimates are added to it

```
    function _burnKLP(
        address asset,
        uint256 klpBurnAmount,
        uint256 klpPrice,
        uint256 klpPricePrecision,
        address ktxVault,
        IRewardRouter rewardRouter
    ) internal returns (uint256) {
        uint256 ktxVaultBalance = IERC20Metadata(asset).balanceOf(ktxVault);
        uint256 maxBurnableKLPAmount = _convertToUSDG(ktxVault, asset, ktxVaultBalance) *
↪   klpPricePrecision / klpPrice;
        uint256 klpAmount = Math.min(klpBurnAmount, maxBurnableKLPAmount);
        uint256 klpRedeemFee =
            _counterKtxFee(ktxVault, asset, klpAmount, klpBurnAmount * klpPrice / KLP_PRICE_PRECISION);
        uint256 counterSwapFee = _counterSwapFee(asset, klpAmount);
=>      uint256 klpAmountWithFee = klpAmount + klpRedeemFee + counterSwapFee;
```

But this estimation is incorrect as:

1. Both calculations use klpAmount itself instead of considering the intermediate value Eg: klpAmount = 100, both fees 5% According to calculation here, totalFee = 100 * 100/95 - 100 + 100 * 100/95 - 100 == 10.526315789 hence totalKLPAmount = 110.526315789 when redeemed, value = 110.526315789 * 0.95 * 0.95 = 99.75 instead of 100
2. The usdg amount used for the redeem fee calculation is `klpBurnAmount * klpPrice / KLP_PRICE_PRECISION`. This is not correct as to get the exact value, the inverse of the `getSellUsdgFeeBasisPoints` function will have to be considered. In case the `klpBurnAmount` differs a lot of from `klpAmount`, the difference could be substantial

2.3.2 Inconsistent price calculation inside _convertToUSDG may result in incorrect price in future

Description: The exchange rate b/w USDG and asset is calculated by using the `IVault(vault).getMaxPrice(asset)` function. Although this is the same function used to determine the exchange rate at the time of liquidity removal, the configuration used will be different ie. this call inside the `_convertToUSDG` function will eventually make a call to the underlying `priceFeed` with the `useSwapPricing` parameter as false while the acutal liquidity removal will set this to `true`

link

```
    function _convertToUSDG(address vault, address asset, uint256 amount) internal view returns
↪   (uint256) {
        /// KTX return price based on chainlink price feed with some customizable logic (e.g USDC
↪   always have 1 to 1 ratio with USDG )
=>      uint256 ktxUsdRate = IVault(vault).getMaxPrice(asset);
        return amount * ktxUsdRate / 10 ** (KLP_DECIMAL - (USDG_DECIMAL -
↪   IERC20Metadata(asset).decimals()));
    }
```

`useSwapPricing` is set to true at the time of actual liquidity removal

```
    function sellUSDG(
        address _token,
        address _receiver
    ) external override nonReentrant returns (uint256) {
        _validateManager();
        _validate(whitelistedTokens[_token], 19);
=>      useSwapPricing = true;

        uint256 usdgAmount = _transferIn(usdg);
        _validate(usdgAmount > 0, 20);

        updateCumulativeFundingRate(_token, _token);

        uint256 redemptionAmount = getRedemptionAmount(_token, usdgAmount);
        _validate(redemptionAmount > 0, 21);
```

The current price feed ignores this variable and hence will not causes any inconsistency. But the vault contract has a setter for the priceFeed which indicate possible update in future

2.4 Informational

2.4.1 STALENESS_THRESHOLD of 24hr can falsely take price feed as stale

The STALENESS_THRESHOLD is hardcoded to 24 hr. But for some pairs (eg: USDC/USD) the heartbet in chainlink is 24 hours. Hence unless the threshold deviation gets breached, the price will only be updated after 24 hours. This can cause some transactions to the LiquidityWarehouse to fail if the request occurs within the time gap of (24 hour + 1 to Chainlink feed update time). But since the price update can be expected to happen soon after the 24 hours, the timeframe will be limited to this

2.4.2 KLPManager has cooldown period which disables withdrawals

Description: The KLPManager contract has a cooldown period during which assets cannot be withdrawn

```
    function _removeLiquidity(
        address _account,
        address _tokenOut,
        uint256 _klpAmount,
        uint256 _minOut,
        address _receiver
    ) private returns (uint256) {
        require(_klpAmount > 0, "KlpManager: invalid _klpAmount");
=>      require(
            lastAddedAt[_account].add(cooldownDuration) <= block.timestamp,
            "KlpManager: cooldown duration not yet passed"
        );
```

Currently this is set to 5 mins. Hence if a deposit has been made to KTX, then the next withdrawal from the user will have to wait 5 mins if any asset is to be removed from the KTX protocol (vs withdrawing fully from the copra contract itself)

2.4.3 KTX LP token is an unverfied contract

Description: The protocol integrates with KTX Finance which uses KLP token to account. But the KLP token contract is unverified hence making it less trustworthy and possible insider exploitation by the team at KTX

link

2.4.4 Critical price adjustments are used in KTX

Description: The VaultPriceFeed of KTX makes critical price adjustements. For eg: if a token is a stable coin, the fetching getMaxPrice() will always return the price as 1 usd no matter how low the token price goes

```
    function getPriceV2(address _token, bool _maximise, bool _includeAmmPrice) public view returns
↪   (uint256) {
        uint256 price = getPrimaryPrice(_token, _maximise);

        if (_includeAmmPrice && isAmmEnabled) {
            price = getAmmPriceV2(_token, _maximise, price);
        }

        if (isSecondaryPriceEnabled) {
            price = getSecondaryPrice(_token, price, _maximise);
        }

        if (strictStableTokens[_token]) {
            uint256 delta = price > ONE_USD ? price.sub(ONE_USD) : ONE_USD.sub(price);
            if (delta <= maxStrictPriceDeviation) {
                return ONE_USD;
            }

            // if _maximise and price is e.g. 1.02, return 1.02
            if (_maximise && price > ONE_USD) {
                return price;
            }

            // if !_maximise and price is e.g. 0.98, return 0.98
            if (!_maximise && price < ONE_USD) {
                return price;
            }

=>          return ONE_USD;
        }
```

This same price will be used in the calculations of the liquidityWarehouse