verichains

SECURITY AUDIT OF

DPEG PROTOCOL SMART CONTRACTS

Public Report

Feb 10, 2023

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Feb 10, 2023. We would like to thank the Vortex for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Dpeg protocol Smart Contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Dpeg protocol Smart Contracts

The Dpeg protocol is a lending and leverage trading protocol enabling traders to short the pegged-coins efficiently and automatically. The Dpeg protocol leverage the trader's position by borrowing the pegged coins provided by staking investors. Thanks to the design of the rebalancing mechanism, the leverage pool will automatically adjust the leverage ratio dynamically. Thus, making the Dpeg protocol solvent without liquidation risk and providing a very user-friendly experience to the investor. Additionally, investors who preferred to stake their assets on the protocol are being offered a much safer product compared to other leveraged protocols within the DeFi ecosystem.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of Dpeg protocol Smart Contracts.

It was conducted on the source code provided by the Vortex team.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
a82615caeb666fa24315e6f942ee9b0afee6918d4c9beb264c7545931d013113	ConfiguratorProxy.sol
a120d78bc0653cfa05a5aeaef90a0c497f301ebcd537f2a9b15ff9f0d157a812	CurveContractInterface.sol
e81846d9d3f8f5c9e8f7ee24f4aae05d3557c2a28141179aef09b67cccfabfa3	CurveSwap.sol
b887e312aa29dcd8a168eee010e5310ee9aae01571836d4d790efc320ea85a1e	DepErc20.sol
666bdceafbd1515e65010dec77abb465123d66446d022f0866850e47d934e76b	DepToken.sol
7a0ce2b184b2efad9a26c39d9ad679e30f05790be2b40741e7b864876901a659	DepTokenInterfaces.sol
cc45a27918c9abbe6b6a8cf72b6f2a32fb85f69f5cc75dbfc81c036547ec1e5f	DepositWithdraw.sol

3c9c70c664c78e159df3d9ee73dacacb5946f7165bf8b6d9d2fc8ad892df7098	DepositWithdrawInterface.sol
a6417e2f1b068faed454627d277dea622483c66d7a863ea7866be80aa979aa70	EIP20Interface.sol
426b23167d8399e5cd8c209147ba3f1492cea11fe4eb5b813b39b8204b3d2c56	EIP20NonStandardInterface.sol
59341efe9dddc152958730a6cef086ce0e5d9cbbd191d9850175273a2b3f218d	ErrorReporter.sol
dd272760b2d9082db9c94f6198896402fe3681bf41fc3f9d8bba31ada290c3c5	ExponentialNoError.sol
4a7bbf4f6d457072ed84bbbe482a3e5b2b1c2fc424946e905cd76a2244d800f8	InterestRateModel.sol
77650ac1819ebe7c30e0ce4fcb32f582a27066df1ababe8a4a58f73046f27fa7	LevErc20.sol
a9d49b9dd151f7839528f2006cce593fb30a98f5c86c588c444be4ab55a298cb	LevToken.sol
3015f7c5ca2c911b31eccd31233663ee7a71d3518dcddb3ccf72a40f5958f33a	LevTokenInterfaces.sol
f2ff8858038e79efd15db2759ed7dad2e2e1232e828a1d2504208bec960931f4	Matrixpricer.sol
8bdbccf66766e9abd0fe0dcc0381e547b6c603cb92e8de0b8d2edc0256e5fa9e	MatrixpricerInterface.sol
7e4177a6523463957a0efdf5295f29cf94bfb3cc367bf4bdfee54400e690172f	MatrixpricerProxyAdmin.sol
a96832ef20e5f1e7b4b11e9ea27e6f40374ecd5ce6e48a63502b99c6e880eaa1	MatrixpricerStorage.sol
e51c12813428b4c0b76753bab3fd5ae7bee69e7df1fc53f344a367edc64840ee	PriceOracle.sol
151153a9d281f6075232e56c74252777f678bb9e7fd4215e6970a9f909cd29c4	PriceOracleLev.sol
1cc9b20e3be772207e6773e5a8b3c77962aa93a3318d847b956e3b800cc44ecc	SimplePriceFeed.sol

393ac908437fd9b37a6d25b30eaaeee1818ef1eb64d7dec79cb42075377866dc	SimplePriceOracleLev.sol
157298b18a38fdf313fb9d981b79118f6008f096d6a7b267f8a8bf9ef404dee5	Tensorpricer.sol
d80dd4ae228af1fd292fd03cd2d757661854d61121e3b92f70c9fb1068f2cd5d	TensorpricerInterface.sol
b735ddcb8930efcc6b61f60f372a8a78a5679627e7634330b4575cf1e6ea52ca	TensorpricerStorage.sol
81c3cc32b63f75b18afa314eb60267486164cae6f0bb5f1dd0965a9f83b78116	V1InterestRateModel.sol

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Dpeg protocol Smart Contracts was written in `Solidity` language, with the required version to be `^0.8.10`. The source code was written based on OpenZeppelin's library.

2.1.1. Depeg contracts

Dpeg is divided into two parts, the first part is to earn fixed income; the second part is to short tokens with leverage.

In the first part, liquidity providers can deposit USDT into the lending pool in exchange for dTokens and earn interest. The deposited coins will be used to lend to Dpeg investors and locked in the leveraged pool. When the demand for de-pegging is low, deposits will be put on other Defi platforms (such as Compound) to earn an income. The interest earned by liquidity providers mainly comes from borrow costs of shorters (levSToken holders).

In the second part, shorters purchase levTokens USDC, and the leverage pool will automatically adjust the leverage according to the market price of USDT/USDC and affect the price of leveraged tokens. Shorters obtain benefits through the price fluctuations of levTokens. Theoretically, shorters will not be liquidated, so the funds of liquidity providers are relatively safe.

2.1.2. levToken

LevToken is an investment portfolio. Considering a short USDT position, LevSUSDT token means short USDT long USDC, denominated in USDC. The portfolio regularly receives the price of USDT/USDC on the Chainlink Oracle and decides whether to rebalance according to the price. The whole process is performed automatically by the smart contract. The price of LevSUSDT will rise as USDT depreciates. Investors obtain leveraged short USDT positions by holding LevSUSDT. In the worst case, the token price falls to 0. The leverage will not be liquidated.

Risk: LevSUSDT will borrow USDT from the staking pool. If the borrowed amount is insufficient due to redemption by staking investors, LevSUSDT will passively return USDT and reduce leverage.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of Dpeg protocol Smart Contracts.

2.2.1. User can redeem the number of tokens less than `exchangeRate` without balance subtraction CRITICAL

The `redeem` function allows the `user` to withdraw tokens through one of 2 parameters (`redeemToken` and `redeemAmount`), the contract will use the `exchangeRate` and the user input to calculate the remaining parameter.

When users pass `redeemAmount` which is less than the `exchangeRate` value, the `redeemToken` is calculated from `redeemTokens = div_(redeemAmountIn, exchangeRate);` statement round to `0`. With the statements after that, the user balance does not decrease but the contract still sends the user `redeemAmount` tokens.

```
//DepErc20.sol
    function redeem(uint redeemTokens, uint redeemAmount) override external returns (uint)
{
        redeemInternal(redeemTokens, redeemAmount);
        return NO_ERROR;
    }
//DepToken.sol
    function redeemInternal(uint redeemTokensIn, uint redeemAmountIn) internal nonReentrant
{
            accrueInterest();
            require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of redeemTokensIn or
redeemAmountIn must be zero");
            Exp memory exchangeRate = Exp({mantissa: exchangeRateStoredInternal() });

            uint redeemTokens;
            uint redeemAmount;
            if (redeemTokensIn > 0) {
                redeemTokens = redeemTokensIn;
                redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTokensIn);
            } else {
                redeemTokens = div_(redeemAmountIn, exchangeRate);
                redeemAmount = redeemAmountIn;
            }

            address payable redeemer = payable(msg.sender);

            uint allowed = matrixpricer.redeemAllowed(address(this), redeemer,
redeemTokens);
            if (allowed != 0) {
                revert RedeemMatrixpricerRejection(allowed);
            }

            if (accrualBlockNumber != getBlockNumber()) {
                revert RedeemFreshnessCheck();
            }

            uint compoundBalance = getCmpBalanceInternal();
```

```
            uint currUSDTBalance = getCashExReserves();
            uint256 totalUndUnborrowed = currUSDTBalance + compoundBalance;
            uint netForceRepayAmt = 0;
            if (totalUndUnborrowed < redeemAmount) {
                uint forceRepayAmtRequest = redeemAmount - totalUndUnborrowed;
                netForceRepayAmt = levErc20.forceRepay(forceRepayAmtRequest);
                if (netForceRepayAmt == 0) {
                    revert RedeemTransferOutNotPossible();
                }else{
                    updateBorrowLedger(netForceRepayAmt, false, true);
                }
                withdrawUSDTfromCmp(compoundBalance);
            }else if (redeemAmount > currUSDTBalance) {
                uint amtNeeded = redeemAmount - currUSDTBalance;
                if(compoundBalance > (amtNeeded + extraUSDT)){
                    withdrawUSDTfromCmp(amtNeeded + extraUSDT);
                }else{
                    withdrawUSDTfromCmp(compoundBalance);
                }
            }
            uint cashAvailToWithdraw = getCashExReserves();
            if(redeemAmount > cashAvailToWithdraw){
                redeemAmount = cashAvailToWithdraw;
            }
            totalSupply = totalSupply - redeemTokens;
            accountTokens[redeemer] = accountTokens[redeemer] - redeemTokens;
            doTransferOut(redeemer, redeemAmount);

            emit Transfer(redeemer, address(this), redeemTokens);
            emit Redeem(redeemer, redeemAmount, redeemTokens);
    }
```

RECOMMENDATION

Add an if-statement like the below to the `redeemInternal` function:

```
function redeemInternal(uint redeemTokensIn, uint redeemAmountIn) internal nonReentrant {
    ...
    if (redeemTokensIn > 0) {
    /*
     * We calculate the exchange rate and the amount of underlying to be redeemed:
     *  redeemTokens = redeemTokensIn
     *  redeemAmount = redeemTokensIn x exchangeRateCurrent
     */
    redeemTokens = redeemTokensIn;
    redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTokensIn);
    } else {
    /*
     * We get the current exchange rate and calculate the amount to be redeemed:
     *  redeemTokens = redeemAmountIn / exchangeRate
     *  redeemAmount = redeemAmountIn
```

```
     */
    redeemTokens = div_(redeemAmountIn, exchangeRate);
    redeemAmount = redeemAmountIn;
    }

    if (redeemTokens == 0 && redeemAmount > 0) {
        revert("redeemTokens zero");
    }
    ...
}
```

UPDATES

- *Dec 26, 2022*: This issue has been acknowledged and fixed by the Vortex team.

2.2.2. LevErc20 - Anyone can call initialize to steal owner role CRITICAL

The `initialize` function is intended to be used only when creating contract data for the first time. However, it is not currently disabled after its initial use, which means that anyone can call it again and potentially gain the owner role and reset the initial data of the contract.

```
//LevErc20.sol
function initialize(address underlying_,
                    address borrowUnderlying_,
                    TensorpricerInterface tensorpricer_,
                    string memory name_,
                    string memory symbol_,
                    uint8 decimals_) public override{
    // LevToken initialize does the bulk of the work
    admin = payable(msg.sender);
    super.initialize(underlying_, borrowUnderlying_, tensorpricer_, name_, symbol_,
decimals_);

    // Set underlying and sanity check it
    underlying = underlying_;
    EIP20Interface(underlying).totalSupply();

    // Set borrow underlying and sanity check it
    borrowUnderlying = borrowUnderlying_;
    EIP20Interface(borrowUnderlying).totalSupply();

    netAssetValue = initialNetAssetValueMantissa;
    hisHighNav = initialNetAssetValueMantissa;
}
//LevToken.sol
function initialize(address underlying_,
    address borrowUnderlying_,
    TensorpricerInterface tensorpricer_,
    string memory name_,
```

```
    string memory symbol_,
    uint8 decimals_) public virtual{
    require(msg.sender == admin, "only admin may initialize the market");

    // Set the tensorpricer
    uint err = _setTensorpricer(tensorpricer_);
    require(err == NO_ERROR, "setting tensorpricer failed");

    name = name_;
    symbol = symbol_;
    decimals = decimals_;

    // The counter starts true to prevent changing it from zero to non-zero (i.e. smaller
cost/refund)
    _notEntered = true;
}
```

RECOMMENDATION

- Disable `initialize` function after the first call.

UPDATES

- *Dec 26, 2022*: This issue has been acknowledged and fixed by the Vortex team.

2.2.3. DepToken - User loses tokens in redeeming process when the balance in the contract is not enough HIGH

The `redeem` function allows users to redeem their `USDT` from `DepToken` contract. But if the `cashAvailToWithdraw` state is less than the `redeemAmount` specified by the user, the contract will only `transferOut` the `cashAvailToWithdraw` value while the `user`'s balance is subtracted by `redeemAmount`.

```
function redeemInternal(uint redeemTokensIn, uint redeemAmountIn) internal nonReentrant {
        accrueInterest();
        require(redeemTokensIn > 0 && redeemAmountIn == 0, "do not support
redeemAmountIn");
        Exp memory exchangeRate = Exp({mantissa: exchangeRateStoredInternal() });

        uint redeemTokens = redeemTokensIn;
        uint redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTokensIn);
        address payable redeemer = payable(msg.sender);
        uint allowed = matrixpricer.redeemAllowed(address(this), redeemer, redeemTokens);
        if (allowed != 0) {
            revert RedeemMatrixpricerRejection(allowed);
        }
        if (accrualBlockNumber != getBlockNumber()) {
            revert RedeemFreshnessCheck();
```

```
        }
        uint compoundBalance = getCmpBalanceInternal();
        uint currUSDTBalance = getCashExReserves();
        uint256 totalUndUnborrowed = currUSDTBalance + compoundBalance;
        uint netForceRepayAmt = 0;

        if (totalUndUnborrowed < redeemAmount) {
            uint forceRepayAmtRequest = redeemAmount - totalUndUnborrowed;
            netForceRepayAmt = levErc20.forceRepay(forceRepayAmtRequest);
            if (netForceRepayAmt == 0) {
                revert RedeemTransferOutNotPossible();
            }else{
                updateBorrowLedger(netForceRepayAmt, false, true);
            }
            withdrawUSDTfromCmp(compoundBalance);
        }else if (redeemAmount > currUSDTBalance) {
            uint amtNeeded = redeemAmount - currUSDTBalance;
            if(compoundBalance > (amtNeeded + extraUSDT)){
                withdrawUSDTfromCmp(amtNeeded + extraUSDT);
            }else{
                withdrawUSDTfromCmp(compoundBalance);
            }
        }
        uint cashAvailToWithdraw = getCashExReserves();
        if(redeemAmount > cashAvailToWithdraw){
            redeemAmount = cashAvailToWithdraw; //<-- only transfer `cashAvailToWithdraw` when balance is not enough
        }
        totalSupply = totalSupply - redeemTokens;
        accountTokens[redeemer] = accountTokens[redeemer] - redeemTokens; //<-- still subtract user's input

        doTransferOut(redeemer, redeemAmount);

        emit Transfer(redeemer, address(this), redeemTokens);
        emit Redeem(redeemer, redeemAmount, redeemTokens, supplyRatePerBlock());
    }
```

RECOMMENDATION

- The contract should revert transaction and keep user's state when the balance is not enough.

UPDATES

- *Dec 26, 2022*: This issue has been acknowledged by the Vortex team.

2.2.4. LevToken - User loses USDC token in minting process when balanceUSDCExReserves is less than mul_(borrowBalanceUSDT, fx_USDTUSDC) HIGH

The volatility of USDC can sometimes cause sudden increases in the value of fx_USDTUSC. In order to mitigate this risk, the LevToken contract includes a function called deleverageAll which is triggered when this occurs. This function swaps all USDC in the contract for USDT and repays these tokens to the DepegToken contract. However, it is important to note that there is a potential vulnerability in which USDC tokens transferred to the contract by users during this process is also swapped without being recorded or refunded to the user.

```
function mintInternal(uint mintAmount) internal nonReentrant {
        address minter = msg.sender;

        uint allowed = tensorpricer.mintAllowed(address(this), minter);
        if (allowed != 0) {
            revert MintTensorpricerRejection(allowed);
        }

        uint fx_USDTUSDC_Mantissa = tensorpricer.getFx('USDTUSDC');
        Exp memory fx_USDTUSDC = Exp({mantissa: fx_USDTUSDC_Mantissa});
        if(!checkLeveragibility(fx_USDTUSDC, mintAmount)){
            return;
        }
        uint actualMintAmount = doTransferIn(minter, mintAmount); //User's USDC is
transferred to the contract

        uint mintTokens;
        uint navAfterTradeMantissa;
        // update a tmp new nav
        Exp memory tmpNav;
        uint tmpBorrowBalanceUSDC = mul_(borrowBalanceUSDT, fx_USDTUSDC);
        uint tmpNavMantissa = calcNetAssetValue(tmpBorrowBalanceUSDC, actualMintAmount);
        bool skipRebalance = false;
        if(tmpNavMantissa == 0){
            if(deleverageAll()){//<-- swap all USDC in contract to USDT and transfer to
DepegToken contract (including the USDC user just sent to the contract)
                return;
            }
            skipRebalance = true;
        }
        ...
function calcNetAssetValue(uint latestBorrowBalanceUSDC, uint offset) internal view returns
(uint){
        uint _totalSupply = totalSupply;
        if (_totalSupply == 0) {
            return initialNetAssetValueMantissa;
        } else {
            uint balanceUSDCExReserves = getCashExReserves() + getCmpBalanceInternal();
            if(balanceUSDCExReserves > latestBorrowBalanceUSDC + offset){
```

```
                return (balanceUSDCExReserves - latestBorrowBalanceUSDC - offset) *
expScale / _totalSupply;
            }else{
                return 0; //<trigger deleverageAll when the balanceUSDCExReserves is not
enough
            }
        }
    }

    function deleverageAll() internal returns (bool) {
    uint cashOnBook = getCashExReserves();
    uint compoundBalance = getCmpBalanceInternal();
    if(compoundBalance > 0) {
        withdrawUSDCfromCmp(compoundBalance)
    }
    cashOnBook = getCashExReserves();
    if(cashOnBook > 0){
        uint finalRepayAmount = changeUSDC2USDT(cashOnBook, 0, address(depErc20));
        uint transFx = cashOnBook * expScale / finalRepayAmount;
        uint origBorrowBalanceUSDT = depErc20.getTotalBorrowsAfterAccrueInterest();
        extraBorrowDemand = 0;
        extraBorrowSupply = 0;
        if(origBorrowBalanceUSDT > finalRepayAmount){
            updateBorrowBalances(transFx, origBorrowBalanceUSDT - finalRepayAmount);
            depErc20.repayBorrow(finalRepayAmount, true);
            updateStats(false, 0, 0, 0);
            tensorpricer._setMintPausedLev(address(this), true);
            tensorpricer._setRedeemPausedLev(address(this), true);
        }else{
            updateBorrowBalances(transFx, 0);
            depErc20.repayBorrow(finalRepayAmount, false);
            updateStats(true, 0, 0, 0);
            return false;
        }
    }
    return true;
}
```

RECOMMENDATION

Refund the tokens user just sent to the contract in this process before swapping all USDC tokens.

UPDATES

- *Dec 26, 2022*: This issue has been acknowledged by the Vortex team.

2.2.5. V1InterestRateModel.sol - The formula in contract is not matched with document

HIGH

The formula in `V1InterestRateModel` contract is not matched with document.

First, the Borrowing Rate is calculated from two explicit variables (`Compoundsupply rate` and `Compound interest rate`) . However, in the source code, the `cRate` state is used for both of these values.

When $0\% < IUR < 80\%$:

$$Borrowing\ Rate = Floor\ rate + 0.8 * Compoundsupply\ rate * IUR$$

$$Floor\ rate = 1.1 * Compound\ interest\ rate$$

```
function getBorrowRate(uint iur, uint cRatePerBlock) override public view returns (uint) {
        uint cRate = cRatePerBlock*blocksPerYear;
        uint borrowRate;
        uint grad = (cRate * 8e17) / MANTISSA;  // *0.8
        if(iur<8e17){    // 0.8
            borrowRate = (11e17 * cRate + iur * grad) / MANTISSA; //<-- Corresponding
(11e17 * cRate + (cRate * 8e17) * iur / MANTISSA) / MANTISSA
        }else{
            uint base = (11e17 * cRate + 8e17 * grad) / MANTISSA;
            uint grad2 = ((1e18 - base)*MANTISSA) / 2e17;
            borrowRate = base + ((iur-8e17)*grad2) / MANTISSA;
        }

        borrowRate = borrowRate/blocksPerYear;
        if(borrowRate>capRatePerBlock){
            borrowRate = capRatePerBlock;
        }else if(borrowRate<floorRatePerBlock){
            borrowRate = floorRatePerBlock;
        }
        return borrowRate;
    }
```

Second, the document defines the `base rate` equals `floorrate * 0.64 * Compoundsupplyrate`. But in source code, it defines the `base rate` equals `floorrate + 0.64 * Compoundsupplyrate`.

When $0\% < IUR < 80\%$:

$$Borrowing\ Rate = Floor\ rate + 0.8 * Compoundsupply\ rate * IUR$$

$$Floor\ rate = 1.1 * Compound\ interest\ rate$$

When $80\% < IUR < 100\%$:

$$BorrowingRate = base\ rate + \frac{1 - base\ rate}{0.2 * (IUR - 0.8)}, base\ rate = floor\ rate * 0.64 * Compoundsupply\ rate$$

```
function getBorrowRate(uint iur, uint cRatePerBlock) override public view returns (uint) {
        uint cRate = cRatePerBlock*blocksPerYear;
        uint borrowRate;
        uint grad = (cRate * 8e17) / MANTISSA;  // *0.8
        if(iur<8e17){    // 0.8
            borrowRate = (11e17 * cRate + iur * grad) / MANTISSA;
        }else{
            uint base = (11e17 * cRate + 8e17 * grad) / MANTISSA;  //<-- Corresponding
(floorrate + 0.64 * Compoundsupplyrate)
            uint grad2 = ((1e18 - base)*MANTISSA) / 2e17;
            borrowRate = base + ((iur-8e17)*grad2) / MANTISSA;
        }

        borrowRate = borrowRate/blocksPerYear;
        if(borrowRate>capRatePerBlock){
            borrowRate = capRatePerBlock;
        }else if(borrowRate<floorRatePerBlock){
            borrowRate = floorRatePerBlock;
        }
        return borrowRate;
    }
```

UPDATES

- *Dec 26, 2022*: This issue has been acknowledged and fixed by the Vortex team. The documents were updated.

2.2.6. LevToken - New holders don't have transfer ability MEDIUM

The leverage tokens were created through the minting process. During this process, the `TensorPrice` contract adds users to the `asset` in order to approve them for transferring tokens. This means that if users transfer their tokens to other users, these recipients will not be able to transfer the tokens to anyone else because they have not been approved.

For approval, new holders must call `mintAllowed` function in `TensorPrice` contract to approve themselves for transferring. However, this process is complex and may be difficult for a typical

user to understand. As a result, it is worth considering updating the logic of the leverage token to make it more user-friendly.

```
//LevToken.sol
function transferTokens(address spender, address src, address dst, uint tokens) internal
returns (uint) {
        /* Fail if transfer not allowed */
        uint allowed = tensorpricer.transferAllowed(address(this), src, dst, tokens); //<--
check approval in tensorPricer
        if (allowed != 0) {
            revert TransferTensorpricerRejection(allowed);   // change the name
        }
        ...
    }
//TensorPricer.sol
function transferAllowed(address levToken, address src, address dst, uint transferTokens)
override external returns (uint) {
    // Currently the only consideration is whether or not
    //  the src is allowed to redeem this many tokens
    uint allowed = redeemAllowedInternal(levToken, src, transferTokens); //<-- Check
allowance in redeemAllowedInternal
    if (allowed != uint(Error.NO_ERROR)) {
        return allowed;
    }

    // core updates
    updateDepegSupplyIndex(levToken);
    distributeSupplierDepeg(levToken, src);
    distributeSupplierDepeg(levToken, dst);

    return uint(Error.NO_ERROR);
}

function redeemAllowedInternal(address levToken, address redeemer, uint redeemTokens)
internal view returns (uint) {

    // check if redeemer has sufficient levToken
    Error err = getHypotheticalAccountLiquidityInternal(redeemer, LevToken(levToken),
redeemTokens);
    if (err != Error.NO_ERROR) {
        return uint(err);
    }

    return uint(Error.NO_ERROR);
}
function getHypotheticalAccountLiquidity(
    address account,
    address levTokenModify,
    uint redeemTokens) public view returns (uint) {
    Error err = getHypotheticalAccountLiquidityInternal(account, LevToken(levTokenModify),
```

```
redeemTokens);
    return uint(err);
}

function getHypotheticalAccountLiquidityInternal(
    address account,
    LevToken levTokenModify,
    uint redeemTokens) internal view returns (Error) {
    AccountLiquidityLocalVars memory vars;
    LevToken[] memory assets = accountAssets[account]; //<-- New holder doesn't have asset
data
    for (uint i = 0; i < assets.length; i++) {
        LevToken asset = assets[i];
        if(asset == levTokenModify){
            (oErr, vars.levTokenBalance) = asset.getAccountSnapshot(account);
            if (oErr != 0) {
                return Error.SNAPSHOT_ERROR;
            }
            if(vars.levTokenBalance >= redeemTokens){
                return Error.NO_ERROR;
            }else{
                return Error.INSUFFICIENT_LIQUIDITY;
            }
        }
    }

    return Error.INSUFFICIENT_LIQUIDITY; //<-- Return error
}
```

UPDATES

- *Dec 26, 2022*: This issue has been acknowledged and fixed by the Vortex team.

2.2.7. Upgradeable contract MEDIUM

The audit scope includes contracts with the `upgradeable` feature, which allows the deployer to alter the logic. If the `deployer` account is compromised, the hacker may exploit this feature to their advantage.

RECOMMENDATION

We suggest changing all `upgradeable` abstract contract to normal contract.

UPDATES

- *Feb 10, 2023*: This issue has been acknowledged by the Vortex team.

2.2.8. LevToken.sol - Redundant code INFORMATIVE

The return data from the first call to the function `getCashExReserves` in the `deleverageAll` function is not utilized. It may be worth considering its removal as a means of saving gas.

```
function deleverageAll() internal returns (bool) {
//        console.log("deleverageAll triggered!");
    uint cashOnBook = getCashExReserves();
    // need to go to compound to get the USDC
    uint compoundBalance = getCmpBalanceInternal();
    if(compoundBalance > 0) {
        withdrawUSDCfromCmp(compoundBalance);  // taking out all we have
    }
    cashOnBook = getCashExReserves();
    ...
```

UPDATES

- *Feb 10, 2023*: This issue has been acknowledged by the Vortex team.

2.2.9. LevErc20.sol, LevToken - Unused argument INFORMATIVE

The `borrowUnderlying_` argument in `initialize` function in both `LevToken` and `LevErc20` contracts are unused. Considering removing it for gas-saving.

```
//LevToken.sol
function initialize(address underlying_,
                    address borrowUnderlying_,
                    TensorpricerInterface tensorpricer_,
                    string memory name_,
                    string memory symbol_,
                    uint8 decimals_) public virtual onlyInitializing {
    require(msg.sender == admin, "only admin may initialize the market");

    // Set the tensorpricer
    uint err = _setTensorpricer(tensorpricer_);
    require(err == NO_ERROR, "setting tensorpricer failed");

    name = name_;
    symbol = symbol_;
    decimals = decimals_;

    // The counter starts true to prevent changing it from zero to non-zero (i.e. smaller
cost/refund)
    _notEntered = true;
}
//LevErc20.sol
function initialize(address underlying_,
    address borrowUnderlying_,
    TensorpricerInterface tensorpricer_,
```

```
    string memory name_,
    string memory symbol_,
    uint8 decimals_) public override initializer {
    // LevToken initialize does the bulk of the work
    admin = payable(msg.sender);
    super.initialize(underlying_, borrowUnderlying_, tensorpricer_, name_, symbol_,
decimals_);

    // Set underlying and sanity check it
    underlying = underlying_;
    EIP20Interface(underlying).totalSupply();

    // Set borrow underlying and sanity check it
    borrowUnderlying = borrowUnderlying_;
    EIP20Interface(borrowUnderlying).totalSupply();

    netAssetValue = initialNetAssetValueMantissa;
    hisHighNav = initialNetAssetValueMantissa;
}
```

UPDATES

- *Feb 10, 2023*: This issue has been acknowledged by the Vortex team.

2.2.10. Hardcoded Value/String INFORMATIVE

Our audit scope contracts contain several hardcoded values and strings in the source code. While this is not currently a problem, it may present difficulties for future development. Therefore, we suggest replacing these hardcoded values and strings with constants for more flexibility and ease of maintenance.

An example of hardcoded values and strings:

```
function updateLedgerInternal() internal {
        updateBorrowBalances(tensorpricer.getFx('USDTUSDC'), depErc20.getTotalBorrows());
// no need to use transFx even tho traded, becoz no lev mint/redeem
        updateStats(true, 0, 0, 0);
        updateNetAssetValue(borrowBalanceUSDC, 0);
    }
    function doRebalanceExt() public {//nonReentrant {
        checkRebalanceRes memory myRes = checkRebalance(2, 0);
        doRebalance(2, myRes, 0);
    }
```

UPDATES

- *Feb 10, 2023*: This issue has been acknowledged by the Vortex team.

2.2.11. Redundant and unimplemented functions INFORMATIVE

The following functions are unused in the contract:

- setPrologue() (found in LevErc20.sol)
- store() (found in Tensorpricer.sol and Matrixpricers.sol)
- retrieve() (found in Tensorpricer.sol and Matrixpricers.sol)

Additionally, there are some functions that are not implemented in the source code:

- updateDepegSupplyIndex() (found in Tensorpricer.sol and Matrixpricer.sol)
- distributeSupplierDepeg() (found in Tensorpricer.sol and Matrixpricer.sol)

It may be worth considering removing these functions in order to clear the source code.

UPDATES

- *Feb 10, 2023*: This issue has been acknowledged by the Vortex team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Dec 26, 2022*	Private Report	Verichains Lab
1.1	*Feb 10, 2023*	Public Report	Verichains Lab

Table 2. Report versions history