Armors Labs

Tender Finance

Smart Contract Audit

Tender Finance Audit Summary

Project name : Tender Finance Contract

Project address: None

Code URL : https://metis.tokenview.com/cn/address/0x670f22666415A7aE45166151F9aa158BeC7C1549

Code URL : https://metis.tokenview.com/cn/address/0x798752C2cd661b3eA4B7A5b45041fA95AcE3fc02

Code URL : https://metis.tokenview.com/cn/address/0xd186010231790CAe3e2f87Ba4982Bf5827B8819D

Code URL : https://metis.tokenview.com/cn/address/0xB01f3D0F5dD254280aC64C89aFB3363d05b91658

Code URL : https://metis.tokenview.com/cn/address/0x18320599eA58B19B3FE12d383F2969C61C1B43F4

Code URL : https://metis.tokenview.com/cn/address/0x2a0DDDb5783E5Cd27821148eDe1B4c90EA739025

Code URL : https://metis.tokenview.com/cn/address/0x0fB0D26Ef8348c43d9eda482e180D54B0296DB22

Code URL : https://metis.tokenview.com/cn/address/0x08EE3541EEB3ba1d519EF4848D8B2A7d75BCE688

Code URL : https://metis.tokenview.com/cn/address/0xA1377dbB30BFdc548eE8c9d7Fa3693E512dD6288

Commit : None

Project target : Tender Finance Contract Audit

Blockchain : Metis

Test result : PASSED

Audit Info

Audit NO : 0X202206170026

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

Tender Finance Audit

The Tender Finance team asked us to review and audit their Tender Finance contract. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
Tender Finance Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2022-06-17

Audit results

Notices

```
    1.The contract oracle does not use a third party, but uses its own Mock contract.

    2.The project party can control the value of the Token, and can also control
      the token address of the trading market
```

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the Tender Finance contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 3 months from the date of output.

Disclaimer

Armors Labs Reports is not and should not be regarded as an "approval" or "disapproval" of any particular project or team. These reports are not and should not be regarded as indicators of the economy or value of any "product" or "asset" created by any team. Armors do not cover testing or auditing the integration with external contract or services (such as Unicrypt, Uniswap, PancakeSwap etc'...)

Armors Labs Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Armors does not guarantee the safety or functionality of the technology agreed to be analyzed.

Armors Labs postulates that the information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered, deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused. Armors Labs Audits should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

Audited target file

file	md5
CErc20	30e41c155ac65301f2e9dd53dac2182c
CErc20Immutable	07021ee32e97c41eae7c48f313f252e6
CToken	b9d8b7ec56186b39a6cbe5b5aa189495
CTokenInterfaces	c5c81b131f5b7bbf75d0c6f7128e8aaf
CarefulMath	b1f19e9f4ff15ac6673cd0d4ce242b01
Comp	90706c55281f234aad00ad6184db5b58
Comptroller	a09cdea78f34c9887f5a02a7a50eac66
ComptrollerInterface	53cac0656d1b25ad2e4a66b14bc7f3a3
ComptrollerStorage	c325b1d599b7d0d359b274c68b3ff0f2
EIP20Interface	fa93e469fb558e63a43784a83f62fc89
EIP20NonStandardInterface	233b54ba1f055b8c2b9ea1c1dd3608f3
ErrorReporter	4360952ef4d39b9c916546d941dca6d1

file	md5
Exponential	eaf3b583cd84e37d7a28223c88d8a114
InterestRateModel	f8e83b5b683c7150fb53ea27df826cbe
MockPriceOracle	863408f37df69e7e3cb6c1794409d93b
PriceOracle	ce2bb7e49cd2129222a5f6722bb500ff
Unitroller	8aa2074397eeaf3bb97e528c2578fe82

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe

Vulnerability	status
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

```
pragma solidity ^0.5.16;

import "./CToken.sol";

/**
 * @title Compound's CErc20 Contract
 * @notice CTokens which wrap an EIP-20 underlying
 * @author Compound
 */
contract CErc20 is CToken, CErc20Interface {
    /**
     * @notice Initialize the new money market
     * @param underlying_ The address of the underlying asset
     * @param comptroller_ The address of the Comptroller
     * @param interestRateModel_ The address of the interest rate model
     * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
     * @param name_ ERC-20 name of this token
     * @param symbol_ ERC-20 symbol of this token
     * @param decimals_ ERC-20 decimal precision of this token
     */
    function initialize(address underlying_,
        ComptrollerInterface comptroller_,
        InterestRateModel interestRateModel_,
        uint initialExchangeRateMantissa_,
        string memory name_,
        string memory symbol_,
        uint8 decimals_) public {
        // CToken initialize does the bulk of the work
        super.initialize(comptroller_, interestRateModel_, initialExchangeRateMantissa_, name_, symbo

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

    /*** User Interface ***/

    /**
     * @notice Sender supplies assets into the market and receives cTokens in exchange
     * @dev Accrues interest whether or not the operation succeeds, unless reverted
     * @param mintAmount The amount of the underlying asset to supply
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function mint(uint mintAmount) external returns (uint) {
        (uint err,) = mintInternal(mintAmount);
        comptroller.addToMarketExternal(address(this), underlying, msg.sender);
        return err;
    }

    /**
     * @notice Sender redeems cTokens in exchange for the underlying asset
     * @dev Accrues interest whether or not the operation succeeds, unless reverted
     * @param redeemTokens The number of cTokens to redeem into underlying
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function redeem(uint redeemTokens) external returns (uint) {
        return redeemInternal(redeemTokens);
    }

    /**
     * @notice Sender redeems cTokens in exchange for a specified amount of underlying asset
     * @dev Accrues interest whether or not the operation succeeds, unless reverted
     * @param redeemAmount The amount of underlying to redeem
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function redeemUnderlying(uint redeemAmount) external returns (uint) {
        return redeemUnderlyingInternal(redeemAmount);
    }

    /**
      * @notice Sender borrows assets from the protocol to their own address
      * @param borrowAmount The amount of the underlying asset to borrow
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function borrow(uint borrowAmount) external returns (uint) {
        return borrowInternal(borrowAmount);
    }

    /**
     * @notice Sender repays their own borrow
     * @param repayAmount The amount to repay
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function repayBorrow(uint repayAmount) external returns (uint) {
        (uint err,) = repayBorrowInternal(repayAmount);
        return err;
    }

    /**
     * @notice Sender repays a borrow belonging to borrower
     * @param borrower the account with the debt being payed off
     * @param repayAmount The amount to repay
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function repayBorrowBehalf(address borrower, uint repayAmount) external returns (uint) {
        (uint err,) = repayBorrowBehalfInternal(borrower, repayAmount);
        return err;
    }

    /**
     * @notice The sender liquidates the borrowers collateral.
     *  The collateral seized is transferred to the liquidator.
     * @param borrower The borrower of this cToken to be liquidated
     * @param repayAmount The amount of the underlying borrowed asset to repay
     * @param cTokenCollateral The market in which to seize collateral from the borrower
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function liquidateBorrow(address borrower, uint repayAmount, CTokenInterface cTokenCollateral) ex
        (uint err,) = liquidateBorrowInternal(borrower, repayAmount, cTokenCollateral);
        return err;
    }

    /**
     * @notice The sender adds to reserves.
     * @param addAmount The amount fo underlying token to add as reserves
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _addReserves(uint addAmount) external returns (uint) {
        return _addReservesInternal(addAmount);
    }

    /*** Safe Token ***/

    /**
     * @notice Gets balance of this contract in terms of the underlying
     * @dev This excludes the value of the current message, if any
     * @return The quantity of underlying tokens owned by this contract
     */
    function getCashPrior() internal view returns (uint) {
        EIP20Interface token = EIP20Interface(underlying);
        return token.balanceOf(address(this));
    }

    /**
     * @dev Similar to EIP20 transfer, except it handles a False result from `transferFrom` and rever
     *      This will revert due to insufficient balance or insufficient allowance.
     *      This function returns the actual amount received,
     *      which may be less than `amount` if there is a fee attached to the transfer.
     *
     *      Note: This wrapper safely handles non-standard ERC-20 tokens that do not return a value.
     *            See here: https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens
     */
    function doTransferIn(address from, uint amount) internal returns (uint) {
        EIP20NonStandardInterface token = EIP20NonStandardInterface(underlying);
        uint balanceBefore = EIP20Interface(underlying).balanceOf(address(this));
        token.transferFrom(from, address(this), amount);

        bool success;
        assembly {
            switch returndatasize()
            case 0 {                       // This is a non-standard ERC-20
                success := not(0)          // set success to true
            }
            case 32 {                      // This is a compliant ERC-20
                returndatacopy(0, 0, 32)
                success := mload(0)        // Set `success = returndata` of external call
            }
            default {                      // This is an excessively non-compliant ERC-20, revert.
                revert(0, 0)
            }
        }
        require(success, "TOKEN_TRANSFER_IN_FAILED");

        // Calculate the amount that was *actually* transferred
        uint balanceAfter = EIP20Interface(underlying).balanceOf(address(this));
        require(balanceAfter >= balanceBefore, "TOKEN_TRANSFER_IN_OVERFLOW");
        return balanceAfter - balanceBefore;   // underflow already checked above, just subtract
    }

    /**
     * @dev Similar to EIP20 transfer, except it handles a False success from `transfer` and returns
     *      error code rather than reverting. If caller has not called checked protocol's balance, th
     *      insufficient cash held in this contract. If caller has checked protocol's balance prior t
     *      it is >= amount, this should not revert in normal conditions.
     *
     *      Note: This wrapper safely handles non-standard ERC-20 tokens that do not return a value.
     *            See here: https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens
     */
    function doTransferOut(address payable to, uint amount) internal {
        EIP20NonStandardInterface token = EIP20NonStandardInterface(underlying);
        token.transfer(to, amount);

        bool success;
        assembly {
            switch returndatasize()
            case 0 {                      // This is a non-standard ERC-20
                success := not(0)          // set success to true
            }
            case 32 {                     // This is a complaint ERC-20
                returndatacopy(0, 0, 32)
                success := mload(0)        // Set `success = returndata` of external call
            }
            default {                     // This is an excessively non-compliant ERC-20, revert.
                revert(0, 0)
            }
        }
        require(success, "TOKEN_TRANSFER_OUT_FAILED");
    }
}



pragma solidity ^0.5.16;

import "./CErc20.sol";

/**
 * @title Compound's CErc20Immutable Contract
 * @notice CTokens which wrap an EIP-20 underlying and are immutable
 * @author Compound
 */
contract CErc20Immutable is CErc20 {
    /**
     * @notice Construct a new money market
     * @param underlying_ The address of the underlying asset
     * @param comptroller_ The address of the Comptroller
     * @param interestRateModel_ The address of the interest rate model
     * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
     * @param name_ ERC-20 name of this token
     * @param symbol_ ERC-20 symbol of this token
     * @param decimals_ ERC-20 decimal precision of this token
     * @param admin_ Address of the administrator of this token
     */
    constructor(address underlying_,
        ComptrollerInterface comptroller_,
        InterestRateModel interestRateModel_,
        uint initialExchangeRateMantissa_,
        string memory name_,
        string memory symbol_,
        uint8 decimals_,
        address payable admin_) public {
        // Creator of the contract is admin during initialization
        admin = msg.sender;

        // Initialize the market
        initialize(underlying_, comptroller_, interestRateModel_, initialExchangeRateMantissa_, name_

        // Set the proper admin now that initialization is done
        admin = admin_;
    }
}



pragma solidity ^0.5.16;

import "./ComptrollerInterface.sol";
import "./CTokenInterfaces.sol";
import "./ErrorReporter.sol";
import "./Exponential.sol";
import "./EIP20Interface.sol";
import "./EIP20NonStandardInterface.sol";
import "./InterestRateModel.sol";

/**
 * @title Compound's CToken Contract
 * @notice Abstract base for CTokens
 * @author Compound
 */
contract CToken is CTokenInterface, Exponential, TokenErrorReporter {
    /**
     * @notice Initialize the money market
     * @param comptroller_ The address of the Comptroller
     * @param interestRateModel_ The address of the interest rate model
     * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
     * @param name_ EIP-20 name of this token
     * @param symbol_ EIP-20 symbol of this token
     * @param decimals_ EIP-20 decimal precision of this token
     */
    function initialize(ComptrollerInterface comptroller_,
        InterestRateModel interestRateModel_,
        uint initialExchangeRateMantissa_,
        string memory name_,
        string memory symbol_,
        uint8 decimals_) public {
        require(msg.sender == admin, "only admin may initialize the market");
        require(accrualBlockNumber == 0 && borrowIndex == 0, "market may only be initialized once");

        // Set initial exchange rate
        initialExchangeRateMantissa = initialExchangeRateMantissa_;
        require(initialExchangeRateMantissa > 0, "initial exchange rate must be greater than zero.");

        // Set the comptroller
        uint err = _setComptroller(comptroller_);
        require(err == uint(Error.NO_ERROR), "setting comptroller failed");

        // Initialize block number and borrow index (block number mocks depend on comptroller being s
        accrualBlockNumber = getBlockNumber();
        borrowIndex = mantissaOne;

        // Set the interest rate model (depends on block number / borrow index)
        err = _setInterestRateModelFresh(interestRateModel_);
        require(err == uint(Error.NO_ERROR), "setting interest rate model failed");

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

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

    /**
     * @notice Transfer `tokens` tokens from `src` to `dst` by `spender`
     * @dev Called by both `transfer` and `transferFrom` internally
     * @param spender The address of the account performing the transfer
     * @param src The address of the source account
     * @param dst The address of the destination account
     * @param tokens The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transferTokens(address spender, address src, address dst, uint tokens) internal returns
        /* Fail if transfer not allowed */
        uint allowed = comptroller.transferAllowed(address(this), src, dst, tokens);
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.TRANSFER_COMPTROLLER_REJECTION
        }

        /* Do not allow self-transfers */
        if (src == dst) {
            return fail(Error.BAD_INPUT, FailureInfo.TRANSFER_NOT_ALLOWED);
        }

        /* Get the allowance, infinite for the account owner */
        uint startingAllowance = 0;
        if (spender == src) {
            startingAllowance = uint(-1);
        } else {
            startingAllowance = transferAllowances[src][spender];
        }

        /* Do the calculations, checking for {under,over}flow */
        MathError mathErr;
        uint allowanceNew;
        uint srcTokensNew;
        uint dstTokensNew;

        (mathErr, allowanceNew) = subUInt(startingAllowance, tokens);
        if (mathErr != MathError.NO_ERROR) {
            return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_NOT_ALLOWED);
        }

        (mathErr, srcTokensNew) = subUInt(accountTokens[src], tokens);
        if (mathErr != MathError.NO_ERROR) {
            return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_NOT_ENOUGH);
        }

        (mathErr, dstTokensNew) = addUInt(accountTokens[dst], tokens);
        if (mathErr != MathError.NO_ERROR) {
            return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_TOO_MUCH);
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        accountTokens[src] = srcTokensNew;
        accountTokens[dst] = dstTokensNew;

        /* Eat some of the allowance (if necessary) */
        if (startingAllowance != uint(-1)) {
            transferAllowances[src][spender] = allowanceNew;
        }

        /* We emit a Transfer event */
        emit Transfer(src, dst, tokens);

        comptroller.transferVerify(address(this), src, dst, tokens);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Transfer `amount` tokens from `msg.sender` to `dst`
     * @param dst The address of the destination account
     * @param amount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transfer(address dst, uint256 amount) external nonReentrant returns (bool) {
        return transferTokens(msg.sender, msg.sender, dst, amount) == uint(Error.NO_ERROR);
    }
```

```
/**
 * @notice Transfer `amount` tokens from `src` to `dst`
 * @param src The address of the source account
 * @param dst The address of the destination account
 * @param amount The number of tokens to transfer
 * @return Whether or not the transfer succeeded
 */
function transferFrom(address src, address dst, uint256 amount) external nonReentrant returns (bo
    return transferTokens(msg.sender, src, dst, amount) == uint(Error.NO_ERROR);
}

/**
 * @notice Approve `spender` to transfer up to `amount` from `src`
 * @dev This will overwrite the approval amount for `spender`
 *  and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
 * @param spender The address of the account which may transfer tokens
 * @param amount The number of tokens that are approved (-1 means infinite)
 * @return Whether or not the approval succeeded
 */
function approve(address spender, uint256 amount) external returns (bool) {
    address src = msg.sender;
    transferAllowances[src][spender] = amount;
    emit Approval(src, spender, amount);
    return true;
}

/**
 * @notice Get the current allowance from `owner` for `spender`
 * @param owner The address of the account which owns the tokens to be spent
 * @param spender The address of the account which may transfer tokens
 * @return The number of tokens allowed to be spent (-1 means infinite)
 */
function allowance(address owner, address spender) external view returns (uint256) {
    return transferAllowances[owner][spender];
}

/**
 * @notice Get the token balance of the `owner`
 * @param owner The address of the account to query
 * @return The number of tokens owned by `owner`
 */
function balanceOf(address owner) external view returns (uint256) {
    return accountTokens[owner];
}

/**
 * @notice Get the underlying balance of the `owner`
 * @dev This also accrues interest in a transaction
 * @param owner The address of the account to query
 * @return The amount of underlying owned by `owner`
 */
function balanceOfUnderlying(address owner) external returns (uint) {
    Exp memory exchangeRate = Exp({mantissa: exchangeRateCurrent()});
    (MathError mErr, uint balance) = mulScalarTruncate(exchangeRate, accountTokens[owner]);
    require(mErr == MathError.NO_ERROR, "balance could not be calculated");
    return balance;
}

/**
 * @notice Get a snapshot of the account's balances, and the cached exchange rate
 * @dev This is used by comptroller to more efficiently perform liquidity checks.
 * @param account Address of the account to snapshot
 * @return (possible error, token balance, borrow balance, exchange rate mantissa)
 */
function getAccountSnapshot(address account) external view returns (uint, uint, uint, uint) {
    uint cTokenBalance = accountTokens[account];
    uint borrowBalance;
    uint exchangeRateMantissa;

    MathError mErr;

    (mErr, borrowBalance) = borrowBalanceStoredInternal(account);
    if (mErr != MathError.NO_ERROR) {
        return (uint(Error.MATH_ERROR), 0, 0, 0);
    }

    (mErr, exchangeRateMantissa) = exchangeRateStoredInternal();
    if (mErr != MathError.NO_ERROR) {
        return (uint(Error.MATH_ERROR), 0, 0, 0);
    }

    return (uint(Error.NO_ERROR), cTokenBalance, borrowBalance, exchangeRateMantissa);
}

/**
 * @dev Function to simply retrieve block number
 *  This exists mainly for inheriting test contracts to stub this result.
 */
function getBlockNumber() internal view returns (uint) {
    return block.number;
}

/**
 * @notice Returns the current per-block borrow interest rate for this cToken
 * @return The borrow interest rate per block, scaled by 1e18
 */
function borrowRatePerBlock() external view returns (uint) {
    return interestRateModel.getBorrowRate(getCashPrior(), totalBorrows, totalReserves);
}

/**
 * @notice Returns the current per-block supply interest rate for this cToken
 * @return The supply interest rate per block, scaled by 1e18
 */
function supplyRatePerBlock() external view returns (uint) {
    return interestRateModel.getSupplyRate(getCashPrior(), totalBorrows, totalReserves, reserveFa
}

/**
 * @notice Returns the current total borrows plus accrued interest
 * @return The total borrows with interest
 */
function totalBorrowsCurrent() external nonReentrant returns (uint) {
    require(accrueInterest() == uint(Error.NO_ERROR), "accrue interest failed");
    return totalBorrows;
}

/**
 * @notice Accrue interest to updated borrowIndex and then calculate account's borrow balance usi
 * @param account The address whose balance should be calculated after updating borrowIndex
 * @return The calculated balance
 */
function borrowBalanceCurrent(address account) external nonReentrant returns (uint) {
    require(accrueInterest() == uint(Error.NO_ERROR), "accrue interest failed");
    return borrowBalanceStored(account);
}

/**
 * @notice Return the borrow balance of account based on stored data
 * @param account The address whose balance should be calculated
 * @return The calculated balance
 */
function borrowBalanceStored(address account) public view returns (uint) {
    (MathError err, uint result) = borrowBalanceStoredInternal(account);
    require(err == MathError.NO_ERROR, "borrowBalanceStored: borrowBalanceStoredInternal failed")
    return result;
}

/**
 * @notice Return the borrow balance of account based on stored data
 * @param account The address whose balance should be calculated
 * @return (error code, the calculated balance or 0 if error code is non-zero)
 */
function borrowBalanceStoredInternal(address account) internal view returns (MathError, uint) {
    /* Note: we do not assert that the market is up to date */
    MathError mathErr;
    uint principalTimesIndex;
    uint result;

    /* Get borrowBalance and borrowIndex */
    BorrowSnapshot storage borrowSnapshot = accountBorrows[account];

    /* If borrowBalance = 0 then borrowIndex is likely also 0.
     * Rather than failing the calculation with a division by 0, we immediately return 0 in this
     */
    if (borrowSnapshot.principal == 0) {
        return (MathError.NO_ERROR, 0);
    }

    /* Calculate new borrow balance using the interest index:
     *  recentBorrowBalance = borrower.borrowBalance * market.borrowIndex / borrower.borrowIndex
     */
    (mathErr, principalTimesIndex) = mulUInt(borrowSnapshot.principal, borrowIndex);
    if (mathErr != MathError.NO_ERROR) {
        return (mathErr, 0);
    }

    (mathErr, result) = divUInt(principalTimesIndex, borrowSnapshot.interestIndex);
    if (mathErr != MathError.NO_ERROR) {
        return (mathErr, 0);
    }

    return (MathError.NO_ERROR, result);
}

/**
 * @notice Accrue interest then return the up-to-date exchange rate
 * @return Calculated exchange rate scaled by 1e18
 */
function exchangeRateCurrent() public nonReentrant returns (uint) {
    require(accrueInterest() == uint(Error.NO_ERROR), "accrue interest failed");
    return exchangeRateStored();
}

/**
 * @notice Calculates the exchange rate from the underlying to the CToken
 * @dev This function does not accrue interest before calculating the exchange rate
 * @return Calculated exchange rate scaled by 1e18
 */
function exchangeRateStored() public view returns (uint) {
    (MathError err, uint result) = exchangeRateStoredInternal();
    require(err == MathError.NO_ERROR, "exchangeRateStored: exchangeRateStoredInternal failed");
    return result;
}

/**
 * @notice Calculates the exchange rate from the underlying to the CToken
 * @dev This function does not accrue interest before calculating the exchange rate
 * @return (error code, calculated exchange rate scaled by 1e18)
 */
function exchangeRateStoredInternal() internal view returns (MathError, uint) {
    uint _totalSupply = totalSupply;
    if (_totalSupply == 0) {
        /*
         * If there are no tokens minted:
         *  exchangeRate = initialExchangeRate
         */
        return (MathError.NO_ERROR, initialExchangeRateMantissa);
    } else {
        /*
         * Otherwise:
         *  exchangeRate = (totalCash + totalBorrows - totalReserves) / totalSupply
         */
        uint totalCash = getCashPrior();
        uint cashPlusBorrowsMinusReserves;
        Exp memory exchangeRate;
        MathError mathErr;

        (mathErr, cashPlusBorrowsMinusReserves) = addThenSubUInt(totalCash, totalBorrows, totalRe
        if (mathErr != MathError.NO_ERROR) {
            return (mathErr, 0);
        }

        (mathErr, exchangeRate) = getExp(cashPlusBorrowsMinusReserves, _totalSupply);
        if (mathErr != MathError.NO_ERROR) {
            return (mathErr, 0);
        }

        return (MathError.NO_ERROR, exchangeRate.mantissa);
    }
}

/**
 * @notice Get cash balance of this cToken in the underlying asset
 * @return The quantity of underlying asset owned by this contract
 */
function getCash() external view returns (uint) {
    return getCashPrior();
}

/**
 * @notice Applies accrued interest to total borrows and reserves
 * @dev This calculates interest accrued from the last checkpointed block
 *   up to the current block and writes new checkpoint to storage.
 */
function accrueInterest() public returns (uint) {
    /* Remember the initial block number */
    uint currentBlockNumber = getBlockNumber();
    uint accrualBlockNumberPrior = accrualBlockNumber;

    /* Short-circuit accumulating 0 interest */
    if (accrualBlockNumberPrior == currentBlockNumber) {
        return uint(Error.NO_ERROR);
    }

    /* Read the previous values out of storage */
    uint cashPrior = getCashPrior();
    uint borrowsPrior = totalBorrows;
    uint reservesPrior = totalReserves;
    uint borrowIndexPrior = borrowIndex;

    /* Calculate the current borrow interest rate */
    uint borrowRateMantissa = interestRateModel.getBorrowRate(cashPrior, borrowsPrior, reservesPr
    require(borrowRateMantissa <= borrowRateMaxMantissa, "borrow rate is absurdly high");

    /* Calculate the number of blocks elapsed since the last accrual */
    (MathError mathErr, uint blockDelta) = subUInt(currentBlockNumber, accrualBlockNumberPrior);
    require(mathErr == MathError.NO_ERROR, "could not calculate block delta");

    /*
     * Calculate the interest accumulated into borrows and reserves and the new index:
     *  simpleInterestFactor = borrowRate * blockDelta
     *  interestAccumulated = simpleInterestFactor * totalBorrows
     *  totalBorrowsNew = interestAccumulated + totalBorrows
     *  totalReservesNew = interestAccumulated * reserveFactor + totalReserves
     *  borrowIndexNew = simpleInterestFactor * borrowIndex + borrowIndex
     */

    Exp memory simpleInterestFactor;
    uint interestAccumulated;
    uint totalBorrowsNew;
    uint totalReservesNew;
    uint borrowIndexNew;

    (mathErr, simpleInterestFactor) = mulScalar(Exp({mantissa: borrowRateMantissa}), blockDelta);
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CA
    }

    (mathErr, interestAccumulated) = mulScalarTruncate(simpleInterestFactor, borrowsPrior);
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALC
    }

    (mathErr, totalBorrowsNew) = addUInt(interestAccumulated, borrowsPrior);
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULA
    }

    (mathErr, totalReservesNew) = mulScalarTruncateAddUInt(Exp({mantissa: reserveFactorMantissa})
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCUL
    }

    (mathErr, borrowIndexNew) = mulScalarTruncateAddUInt(simpleInterestFactor, borrowIndexPrior,
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULAT
    }

    /////////////////////////
    // EFFECTS & INTERACTIONS
    // (No safe failures beyond this point)

    /* We write the previously calculated values into storage */
    accrualBlockNumber = currentBlockNumber;
    borrowIndex = borrowIndexNew;
    totalBorrows = totalBorrowsNew;
    totalReserves = totalReservesNew;

    /* We emit an AccrueInterest event */
    emit AccrueInterest(cashPrior, interestAccumulated, borrowIndexNew, totalBorrowsNew);

    return uint(Error.NO_ERROR);
}

/**
 * @notice Sender supplies assets into the market and receives cTokens in exchange
 * @dev Accrues interest whether or not the operation succeeds, unless reverted
 * @param mintAmount The amount of the underlying asset to supply
 * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), an
 */
function mintInternal(uint mintAmount) internal nonReentrant returns (uint, uint) {
    uint error = accrueInterest();
    if (error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact that an attempt
        return (fail(Error(error), FailureInfo.MINT_ACCRUE_INTEREST_FAILED), 0);
    }
    // mintFresh emits the actual Mint event if successful and logs on errors, so we don't need t
    return mintFresh(msg.sender, mintAmount);
}

struct MintLocalVars {
    Error err;
    MathError mathErr;
    uint exchangeRateMantissa;
    uint mintTokens;
    uint totalSupplyNew;
    uint accountTokensNew;
    uint actualMintAmount;
}

/**
 * @notice User supplies assets into the market and receives cTokens in exchange
 * @dev Assumes interest has already been accrued up to the current block
 * @param minter The address of the account which is supplying the assets
 * @param mintAmount The amount of the underlying asset to supply
 * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), an
 */
function mintFresh(address minter, uint mintAmount) internal returns (uint, uint) {
    /* Fail if mint not allowed */
    uint allowed = comptroller.mintAllowed(address(this), minter, mintAmount);
    if (allowed != 0) {
        return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.MINT_COMPTROLLER_REJECTION, a
    }

    /* Verify market's block number equals current block number */
    if (accrualBlockNumber != getBlockNumber()) {
        return (fail(Error.MARKET_NOT_FRESH, FailureInfo.MINT_FRESHNESS_CHECK), 0);
    }

    MintLocalVars memory vars;

    (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal();
    if (vars.mathErr != MathError.NO_ERROR) {
        return (failOpaque(Error.MATH_ERROR, FailureInfo.MINT_EXCHANGE_RATE_READ_FAILED, uint(var
    }

    /////////////////////////
    // EFFECTS & INTERACTIONS
    // (No safe failures beyond this point)

    /*
     *  We call `doTransferIn` for the minter and the mintAmount.
     *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
     *  `doTransferIn` reverts if anything goes wrong, since we can't be sure if
     *  side-effects occurred. The function returns the amount actually transferred,
     *  in case of a fee. On success, the cToken holds an additional `actualMintAmount`
     *  of cash.
     */
    vars.actualMintAmount = doTransferIn(minter, mintAmount);

    /*
     * We get the current exchange rate and calculate the number of cTokens to be minted:
     *  mintTokens = actualMintAmount / exchangeRate
     */
    (vars.mathErr, vars.mintTokens) = divScalarByExpTruncate(vars.actualMintAmount, Exp({mantissa
    require(vars.mathErr == MathError.NO_ERROR, "MINT_EXCHANGE_CALCULATION_FAILED");

    /*
     * We calculate the new total supply of cTokens and minter token balance, checking for overfl
     *  totalSupplyNew = totalSupply + mintTokens
     *  accountTokensNew = accountTokens[minter] + mintTokens
     */
    (vars.mathErr, vars.totalSupplyNew) = addUInt(totalSupply, vars.mintTokens);
    require(vars.mathErr == MathError.NO_ERROR, "MINT_NEW_TOTAL_SUPPLY_CALCULATION_FAILED");

    (vars.mathErr, vars.accountTokensNew) = addUInt(accountTokens[minter], vars.mintTokens);
    require(vars.mathErr == MathError.NO_ERROR, "MINT_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED");

    /* We write previously calculated values into storage */
    totalSupply = vars.totalSupplyNew;
    accountTokens[minter] = vars.accountTokensNew;

    /* We emit a Mint event, and a Transfer event */
    emit Mint(minter, vars.actualMintAmount, vars.mintTokens);
    emit Transfer(address(this), minter, vars.mintTokens);

    /* We call the defense hook */
    comptroller.mintVerify(address(this), minter, vars.actualMintAmount, vars.mintTokens);

    return (uint(Error.NO_ERROR), vars.actualMintAmount);
}

/**
 * @notice Sender redeems cTokens in exchange for the underlying asset
 * @dev Accrues interest whether or not the operation succeeds, unless reverted
 * @param redeemTokens The number of cTokens to redeem into underlying
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function redeemInternal(uint redeemTokens) internal nonReentrant returns (uint) {
    uint error = accrueInterest();
    if (error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact that an attempt
        return fail(Error(error), FailureInfo.REDEEM_ACCRUE_INTEREST_FAILED);
    }
    // redeemFresh emits redeem-specific logs on errors, so we don't need to
    return redeemFresh(msg.sender, redeemTokens, 0);
}

/**
 * @notice Sender redeems cTokens in exchange for a specified amount of underlying asset
 * @dev Accrues interest whether or not the operation succeeds, unless reverted
 * @param redeemAmount The amount of underlying to receive from redeeming cTokens
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function redeemUnderlyingInternal(uint redeemAmount) internal nonReentrant returns (uint) {
    uint error = accrueInterest();
    if (error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact that an attempt
        return fail(Error(error), FailureInfo.REDEEM_ACCRUE_INTEREST_FAILED);
    }
    // redeemFresh emits redeem-specific logs on errors, so we don't need to
    return redeemFresh(msg.sender, 0, redeemAmount);
}

struct RedeemLocalVars {
    Error err;
    MathError mathErr;
    uint exchangeRateMantissa;
    uint redeemTokens;
    uint redeemAmount;
    uint totalSupplyNew;
    uint accountTokensNew;
}

/**
 * @notice User redeems cTokens in exchange for the underlying asset
 * @dev Assumes interest has already been accrued up to the current block
 * @param redeemer The address of the account which is redeeming the tokens
 * @param redeemTokensIn The number of cTokens to redeem into underlying (only one of redeemToken
 * @param redeemAmountIn The number of underlying tokens to receive from redeeming cTokens (only
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function redeemFresh(address payable redeemer, uint redeemTokensIn, uint redeemAmountIn) internal
    require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of redeemTokensIn or redeemAmountIn

    RedeemLocalVars memory vars;

    /* exchangeRate = invoke Exchange Rate Stored() */
    (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal();
    if (vars.mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_RATE_READ_FAILED, uint(va
    }

    /* If redeemTokensIn > 0: */
    if (redeemTokensIn > 0) {
        /*
         * We calculate the exchange rate and the amount of underlying to be redeemed:
         *  redeemTokens = redeemTokensIn
         *  redeemAmount = redeemTokensIn x exchangeRateCurrent
         */
        vars.redeemTokens = redeemTokensIn;

        (vars.mathErr, vars.redeemAmount) = mulScalarTruncate(Exp({mantissa: vars.exchangeRateMan
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_TOKENS_CALCULATION_FA
        }
    } else {
        /*
         * We get the current exchange rate and calculate the amount to be redeemed:
         *  redeemTokens = redeemAmountIn / exchangeRate
         *  redeemAmount = redeemAmountIn
         */

        (vars.mathErr, vars.redeemTokens) = divScalarByExpTruncate(redeemAmountIn, Exp({mantissa:
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_AMOUNT_CALCULATION_FA
        }

        vars.redeemAmount = redeemAmountIn;
    }

    /* Fail if redeem not allowed */
    uint allowed = comptroller.redeemAllowed(address(this), redeemer, vars.redeemTokens);
    if (allowed != 0) {
        return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.REDEEM_COMPTROLLER_REJECTION,
    }

    /* Verify market's block number equals current block number */
    if (accrualBlockNumber != getBlockNumber()) {
        return fail(Error.MARKET_NOT_FRESH, FailureInfo.REDEEM_FRESHNESS_CHECK);
    }

    /*
     * We calculate the new total supply and redeemer balance, checking for underflow:
     *  totalSupplyNew = totalSupply - redeemTokens
         *  accountTokensNew = accountTokens[redeemer] - redeemTokens
         */
        (vars.mathErr, vars.totalSupplyNew) = subUInt(totalSupply, vars.redeemTokens);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_NEW_TOTAL_SUPPLY_CALCULATION_FAILE
        }

        (vars.mathErr, vars.accountTokensNew) = subUInt(accountTokens[redeemer], vars.redeemTokens);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_NEW_ACCOUNT_BALANCE_CALCULATION_FA
        }

        /* Fail gracefully if protocol has insufficient cash */
        if (getCashPrior() < vars.redeemAmount) {
            return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.REDEEM_TRANSFER_OUT_NOT_POSSIBLE);
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         * We invoke doTransferOut for the redeemer and the redeemAmount.
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken has redeemAmount less of cash.
         *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occu
         */
        doTransferOut(redeemer, vars.redeemAmount);

        /* We write previously calculated values into storage */
        totalSupply = vars.totalSupplyNew;
        accountTokens[redeemer] = vars.accountTokensNew;

        /* We emit a Transfer event, and a Redeem event */
        emit Transfer(redeemer, address(this), vars.redeemTokens);
        emit Redeem(redeemer, vars.redeemAmount, vars.redeemTokens);

        /* We call the defense hook */
        comptroller.redeemVerify(address(this), redeemer, vars.redeemAmount, vars.redeemTokens);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Sender borrows assets from the protocol to their own address
      * @param borrowAmount The amount of the underlying asset to borrow
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function borrowInternal(uint borrowAmount) internal nonReentrant returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempt
            return fail(Error(error), FailureInfo.BORROW_ACCRUE_INTEREST_FAILED);
        }
        // borrowFresh emits borrow-specific logs on errors, so we don't need to
        return borrowFresh(msg.sender, borrowAmount);
    }

    struct BorrowLocalVars {
        MathError mathErr;
        uint accountBorrows;
        uint accountBorrowsNew;
        uint totalBorrowsNew;
    }

    /**
```

```
     * @notice Users borrow assets from the protocol to their own address
     * @param borrowAmount The amount of the underlying asset to borrow
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function borrowFresh(address payable borrower, uint borrowAmount) internal returns (uint) {
        /* Fail if borrow not allowed */
        uint allowed = comptroller.borrowAllowed(address(this), borrower, borrowAmount);
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.BORROW_COMPTROLLER_REJECTION,
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.BORROW_FRESHNESS_CHECK);
        }

        /* Fail gracefully if protocol has insufficient underlying cash */
        if (getCashPrior() < borrowAmount) {
            return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.BORROW_CASH_NOT_AVAILABLE);
        }

        BorrowLocalVars memory vars;

        /*
         * We calculate the new borrower and total borrow balances, failing on overflow:
         *  accountBorrowsNew = accountBorrows + borrowAmount
         *  totalBorrowsNew = totalBorrows + borrowAmount
         */
        (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.BORROW_ACCUMULATED_BALANCE_CALCULATION_FA
        }

        (vars.mathErr, vars.accountBorrowsNew) = addUInt(vars.accountBorrows, borrowAmount);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULA
        }

        (vars.mathErr, vars.totalBorrowsNew) = addUInt(totalBorrows, borrowAmount);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAIL
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         * We invoke doTransferOut for the borrower and the borrowAmount.
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken borrowAmount less of cash.
         *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occu
         */
        doTransferOut(borrower, borrowAmount);

        /* We write the previously calculated values into storage */
        accountBorrows[borrower].principal = vars.accountBorrowsNew;
        accountBorrows[borrower].interestIndex = borrowIndex;
        totalBorrows = vars.totalBorrowsNew;

        /* We emit a Borrow event */
        emit Borrow(borrower, borrowAmount, vars.accountBorrowsNew, vars.totalBorrowsNew);

        /* We call the defense hook */
        comptroller.borrowVerify(address(this), borrower, borrowAmount);
        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Sender repays their own borrow
     * @param repayAmount The amount to repay
     * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), an
     */
    function repayBorrowInternal(uint repayAmount) internal nonReentrant returns (uint, uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempt
            return (fail(Error(error), FailureInfo.REPAY_BORROW_ACCRUE_INTEREST_FAILED), 0);
        }
        // repayBorrowFresh emits repay-borrow-specific logs on errors, so we don't need to
        return repayBorrowFresh(msg.sender, msg.sender, repayAmount);
    }

    /**
     * @notice Sender repays a borrow belonging to borrower
     * @param borrower the account with the debt being payed off
     * @param repayAmount The amount to repay
     * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), an
     */
    function repayBorrowBehalfInternal(address borrower, uint repayAmount) internal nonReentrant retu
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempt
            return (fail(Error(error), FailureInfo.REPAY_BEHALF_ACCRUE_INTEREST_FAILED), 0);
        }
        // repayBorrowFresh emits repay-borrow-specific logs on errors, so we don't need to
        return repayBorrowFresh(msg.sender, borrower, repayAmount);
    }

    struct RepayBorrowLocalVars {
        Error err;
        MathError mathErr;
        uint repayAmount;
        uint borrowerIndex;
        uint accountBorrows;
        uint accountBorrowsNew;
        uint totalBorrowsNew;
        uint actualRepayAmount;
    }

    /**
     * @notice Borrows are repaid by another user (possibly the borrower).
     * @param payer the account paying off the borrow
     * @param borrower the account with the debt being payed off
     * @param repayAmount the amount of undelrying tokens being returned
     * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), an
     */
    function repayBorrowFresh(address payer, address borrower, uint repayAmount) internal returns (ui
        /* Fail if repayBorrow not allowed */
        uint allowed = comptroller.repayBorrowAllowed(address(this), payer, borrower, repayAmount);
        if (allowed != 0) {
            return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.REPAY_BORROW_COMPTROLLER_REJE
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.REPAY_BORROW_FRESHNESS_CHECK), 0);
        }

        RepayBorrowLocalVars memory vars;
        /* We remember the original borrowerIndex for verification purposes */
        vars.borrowerIndex = accountBorrows[borrower].interestIndex;

        /* We fetch the amount the borrower owes, with accumulated interest */
        (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
        if (vars.mathErr != MathError.NO_ERROR) {
            return (failOpaque(Error.MATH_ERROR, FailureInfo.REPAY_BORROW_ACCUMULATED_BALANCE_CALCULA
        }

        /* If repayAmount == -1, repayAmount = accountBorrows */
        if (repayAmount == uint(-1)) {
            vars.repayAmount = vars.accountBorrows;
        } else {
            vars.repayAmount = repayAmount;
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         * We call doTransferIn for the payer and the repayAmount
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken holds an additional repayAmount of cash.
         *  doTransferIn reverts if anything goes wrong, since we can't be sure if side effects occur
         *   it returns the amount actually transferred, in case of a fee.
         */
        vars.actualRepayAmount = doTransferIn(payer, vars.repayAmount);

        /*
         * We calculate the new borrower and total borrow balances, failing on underflow:
         *  accountBorrowsNew = accountBorrows - actualRepayAmount
         *  totalBorrowsNew = totalBorrows - actualRepayAmount
         */
        (vars.mathErr, vars.accountBorrowsNew) = subUInt(vars.accountBorrows, vars.actualRepayAmount)
        require(vars.mathErr == MathError.NO_ERROR, "REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULAT

        (vars.mathErr, vars.totalBorrowsNew) = subUInt(totalBorrows, vars.actualRepayAmount);
        require(vars.mathErr == MathError.NO_ERROR, "REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILE

        /* We write the previously calculated values into storage */
        accountBorrows[borrower].principal = vars.accountBorrowsNew;
        accountBorrows[borrower].interestIndex = borrowIndex;
        totalBorrows = vars.totalBorrowsNew;

        /* We emit a RepayBorrow event */
        emit RepayBorrow(payer, borrower, vars.actualRepayAmount, vars.accountBorrowsNew, vars.totalB

        /* We call the defense hook */
        comptroller.repayBorrowVerify(address(this), payer, borrower, vars.actualRepayAmount, vars.bo

        return (uint(Error.NO_ERROR), vars.actualRepayAmount);
    }

    /**
     * @notice The sender liquidates the borrowers collateral.
     *  The collateral seized is transferred to the liquidator.
     * @param borrower The borrower of this cToken to be liquidated
     * @param cTokenCollateral The market in which to seize collateral from the borrower
     * @param repayAmount The amount of the underlying borrowed asset to repay
     * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), an
     */
    function liquidateBorrowInternal(address borrower, uint repayAmount, CTokenInterface cTokenCollat
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempt
            return (fail(Error(error), FailureInfo.LIQUIDATE_ACCRUE_BORROW_INTEREST_FAILED), 0);
        }

        error = cTokenCollateral.accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempt
            return (fail(Error(error), FailureInfo.LIQUIDATE_ACCRUE_COLLATERAL_INTEREST_FAILED), 0);
        }

        // liquidateBorrowFresh emits borrow-specific logs on errors, so we don't need to
        return liquidateBorrowFresh(msg.sender, borrower, repayAmount, cTokenCollateral);
    }

    /**
     * @notice The liquidator liquidates the borrowers collateral.
     *  The collateral seized is transferred to the liquidator.
     * @param borrower The borrower of this cToken to be liquidated
     * @param liquidator The address repaying the borrow and seizing collateral
     * @param cTokenCollateral The market in which to seize collateral from the borrower
     * @param repayAmount The amount of the underlying borrowed asset to repay
     * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), an
     */
    function liquidateBorrowFresh(address liquidator, address borrower, uint repayAmount, CTokenInter
        /* Fail if liquidate not allowed */
        uint allowed = comptroller.liquidateBorrowAllowed(address(this), address(cTokenCollateral), l
        if (allowed != 0) {
            return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.LIQUIDATE_COMPTROLLER_REJECTI
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.LIQUIDATE_FRESHNESS_CHECK), 0);
        }

        /* Verify cTokenCollateral market's block number equals current block number */
        if (cTokenCollateral.accrualBlockNumber() != getBlockNumber()) {
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.LIQUIDATE_COLLATERAL_FRESHNESS_CHECK), 0
        }

        /* Fail if borrower = liquidator */
        if (borrower == liquidator) {
            return (fail(Error.INVALID_ACCOUNT_PAIR, FailureInfo.LIQUIDATE_LIQUIDATOR_IS_BORROWER), 0
        }

        /* Fail if repayAmount = 0 */
        if (repayAmount == 0) {
            return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED, FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_
        }

        /* Fail if repayAmount = -1 */
        if (repayAmount == uint(-1)) {
            return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED, FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_
        }


        /* Fail if repayBorrow fails */
        (uint repayBorrowError, uint actualRepayAmount) = repayBorrowFresh(liquidator, borrower, repa
        if (repayBorrowError != uint(Error.NO_ERROR)) {
            return (fail(Error(repayBorrowError), FailureInfo.LIQUIDATE_REPAY_BORROW_FRESH_FAILED), 0
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /* We calculate the number of collateral tokens that will be seized */
        (uint amountSeizeError, uint seizeTokens) = comptroller.liquidateCalculateSeizeTokens(address
        require(amountSeizeError == uint(Error.NO_ERROR), "LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEI

        /* Revert if borrower collateral token balance < seizeTokens */
        require(cTokenCollateral.balanceOf(borrower) >= seizeTokens, "LIQUIDATE_SEIZE_TOO_MUCH");

        // If this is also the collateral, run seizeInternal to avoid re-entrancy, otherwise make an
        uint seizeError;
        if (address(cTokenCollateral) == address(this)) {
            seizeError = seizeInternal(address(this), liquidator, borrower, seizeTokens);
        } else {
            seizeError = cTokenCollateral.seize(liquidator, borrower, seizeTokens);
        }

        /* Revert if seize tokens fails (since we cannot be sure of side effects) */
        require(seizeError == uint(Error.NO_ERROR), "token seizure failed");

        /* We emit a LiquidateBorrow event */
        emit LiquidateBorrow(liquidator, borrower, actualRepayAmount, address(cTokenCollateral), seiz

        /* We call the defense hook */
        comptroller.liquidateBorrowVerify(address(this), address(cTokenCollateral), liquidator, borro

        return (uint(Error.NO_ERROR), actualRepayAmount);
    }

    /**
     * @notice Transfers collateral tokens (this market) to the liquidator.
     * @dev Will fail unless called by another cToken during the process of liquidation.
     *  Its absolutely critical to use msg.sender as the borrowed cToken and not a parameter.
     * @param liquidator The account receiving seized collateral
     * @param borrower The account having collateral seized
     * @param seizeTokens The number of cTokens to seize
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function seize(address liquidator, address borrower, uint seizeTokens) external nonReentrant retu
        return seizeInternal(msg.sender, liquidator, borrower, seizeTokens);
    }

    /**
     * @notice Transfers collateral tokens (this market) to the liquidator.
     * @dev Called only during an in-kind liquidation, or by liquidateBorrow during the liquidation o
     *  Its absolutely critical to use msg.sender as the seizer cToken and not a parameter.
     * @param seizerToken The contract seizing the collateral (i.e. borrowed cToken)
     * @param liquidator The account receiving seized collateral
     * @param borrower The account having collateral seized
     * @param seizeTokens The number of cTokens to seize
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function seizeInternal(address seizerToken, address liquidator, address borrower, uint seizeToken
        /* Fail if seize not allowed */
        uint allowed = comptroller.seizeAllowed(address(this), seizerToken, liquidator, borrower, sei
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.LIQUIDATE_SEIZE_COMPTROLLER_RE
        }

        /* Fail if borrower = liquidator */
        if (borrower == liquidator) {
            return fail(Error.INVALID_ACCOUNT_PAIR, FailureInfo.LIQUIDATE_SEIZE_LIQUIDATOR_IS_BORROWE
        }

        MathError mathErr;
        uint borrowerTokensNew;
        uint liquidatorTokensNew;

        /*
```

```
         * We calculate the new borrower and liquidator token balances, failing on underflow/overflow
         *  borrowerTokensNew = accountTokens[borrower] - seizeTokens
         *  liquidatorTokensNew = accountTokens[liquidator] + seizeTokens
         */
        (mathErr, borrowerTokensNew) = subUInt(accountTokens[borrower], seizeTokens);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.LIQUIDATE_SEIZE_BALANCE_DECREMENT_FAILED,
        }

        (mathErr, liquidatorTokensNew) = addUInt(accountTokens[liquidator], seizeTokens);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED,
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /* We write the previously calculated values into storage */
        accountTokens[borrower] = borrowerTokensNew;
        accountTokens[liquidator] = liquidatorTokensNew;

        /* Emit a Transfer event */
        emit Transfer(borrower, liquidator, seizeTokens);

        /* We call the defense hook */
        comptroller.seizeVerify(address(this), seizerToken, liquidator, borrower, seizeTokens);

        return uint(Error.NO_ERROR);
    }


    /*** Admin Functions ***/

    /**
      * @notice Begins transfer of admin rights. The newPendingAdmin must call `_acceptAdmin` to fina
      * @dev Admin function to begin change of admin. The newPendingAdmin must call `_acceptAdmin` to
      * @param newPendingAdmin New pending admin.
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setPendingAdmin(address payable newPendingAdmin) external returns (uint) {
        // Check caller = admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PENDING_ADMIN_OWNER_CHECK);
        }

        // Save current value, if any, for inclusion in log
        address oldPendingAdmin = pendingAdmin;

        // Store pendingAdmin with value newPendingAdmin
        pendingAdmin = newPendingAdmin;

        // Emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin)
        emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
      * @dev Admin function for pending admin to accept role and update admin
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _acceptAdmin() external returns (uint) {
        // Check caller is pendingAdmin and pendingAdmin ≠ address(0)
        if (msg.sender != pendingAdmin || msg.sender == address(0)) {
            return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_ADMIN_PENDING_ADMIN_CHECK);
        }

        // Save current values for inclusion in log
        address oldAdmin = admin;
        address oldPendingAdmin = pendingAdmin;

        // Store admin with value pendingAdmin
        admin = pendingAdmin;

        // Clear the pending value
        pendingAdmin = address(0);

        emit NewAdmin(oldAdmin, admin);
        emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Sets a new comptroller for the market
      * @dev Admin function to set a new comptroller
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setComptroller(ComptrollerInterface newComptroller) public returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_COMPTROLLER_OWNER_CHECK);
        }

        ComptrollerInterface oldComptroller = comptroller;
        // Ensure invoke comptroller.isComptroller() returns true
        require(newComptroller.isComptroller(), "marker method returned false");

        // Set market's comptroller to newComptroller
        comptroller = newComptroller;

        // Emit NewComptroller(oldComptroller, newComptroller)
        emit NewComptroller(oldComptroller, newComptroller);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice accrues interest and sets a new reserve factor for the protocol using _setReserveFact
      * @dev Admin function to accrue interest and set a new reserve factor
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setReserveFactor(uint newReserveFactorMantissa) external nonReentrant returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but on top of that we want to log the fact that a
            return fail(Error(error), FailureInfo.SET_RESERVE_FACTOR_ACCRUE_INTEREST_FAILED);
        }
        // _setReserveFactorFresh emits reserve-factor-specific logs on errors, so we don't need to.
        return _setReserveFactorFresh(newReserveFactorMantissa);
    }

    /**
      * @notice Sets a new reserve factor for the protocol (*requires fresh interest accrual)
      * @dev Admin function to set a new reserve factor
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setReserveFactorFresh(uint newReserveFactorMantissa) internal returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_RESERVE_FACTOR_ADMIN_CHECK);
        }

        // Verify market's block number equals current block number
        if (accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.SET_RESERVE_FACTOR_FRESH_CHECK);
        }

        // Check newReserveFactor ≤ maxReserveFactor
        if (newReserveFactorMantissa > reserveFactorMaxMantissa) {
            return fail(Error.BAD_INPUT, FailureInfo.SET_RESERVE_FACTOR_BOUNDS_CHECK);
        }

        uint oldReserveFactorMantissa = reserveFactorMantissa;
        reserveFactorMantissa = newReserveFactorMantissa;

        emit NewReserveFactor(oldReserveFactorMantissa, newReserveFactorMantissa);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Accrues interest and reduces reserves by transferring from msg.sender
     * @param addAmount Amount of addition to reserves
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _addReservesInternal(uint addAmount) internal nonReentrant returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but on top of that we want to log the fact that a
            return fail(Error(error), FailureInfo.ADD_RESERVES_ACCRUE_INTEREST_FAILED);
        }

        // _addReservesFresh emits reserve-addition-specific logs on errors, so we don't need to.
        (error, ) = _addReservesFresh(addAmount);
        return error;
    }

    /**
     * @notice Add reserves by transferring from caller
     * @dev Requires fresh interest accrual
     * @param addAmount Amount of addition to reserves
     * @return (uint, uint) An error code (0=success, otherwise a failure (see ErrorReporter.sol for
     */
    function _addReservesFresh(uint addAmount) internal returns (uint, uint) {
        // totalReserves + actualAddAmount
        uint totalReservesNew;
        uint actualAddAmount;

        // We fail gracefully unless market's block number equals current block number
        if (accrualBlockNumber != getBlockNumber()) {
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.ADD_RESERVES_FRESH_CHECK), actualAddAmou
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         * We call doTransferIn for the caller and the addAmount
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken holds an additional addAmount of cash.
         *  doTransferIn reverts if anything goes wrong, since we can't be sure if side effects occur
         *  it returns the amount actually transferred, in case of a fee.
         */
        actualAddAmount = doTransferIn(msg.sender, addAmount);

        totalReservesNew = totalReserves + actualAddAmount;

        /* Revert on overflow */
        require(totalReservesNew >= totalReserves, "add reserves unexpected overflow");

        // Store reserves[n+1] = reserves[n] + actualAddAmount
        totalReserves = totalReservesNew;

        /* Emit NewReserves(admin, actualAddAmount, reserves[n+1]) */
        emit ReservesAdded(msg.sender, actualAddAmount, totalReservesNew);

        /* Return (NO_ERROR, actualAddAmount) */
        return (uint(Error.NO_ERROR), actualAddAmount);
    }


    /**
     * @notice Accrues interest and reduces reserves by transferring to admin
     * @param reduceAmount Amount of reduction to reserves
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _reduceReserves(uint reduceAmount) external nonReentrant returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but on top of that we want to log the fact that a
            return fail(Error(error), FailureInfo.REDUCE_RESERVES_ACCRUE_INTEREST_FAILED);
        }
        // _reduceReservesFresh emits reserve-reduction-specific logs on errors, so we don't need to.
        return _reduceReservesFresh(reduceAmount);
    }

    /**
     * @notice Reduces reserves by transferring to admin
     * @dev Requires fresh interest accrual
     * @param reduceAmount Amount of reduction to reserves
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _reduceReservesFresh(uint reduceAmount) internal returns (uint) {
        // totalReserves - reduceAmount
        uint totalReservesNew;

        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.REDUCE_RESERVES_ADMIN_CHECK);
        }

        // We fail gracefully unless market's block number equals current block number
        if (accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.REDUCE_RESERVES_FRESH_CHECK);
        }

        // Fail gracefully if protocol has insufficient underlying cash
        if (getCashPrior() < reduceAmount) {
            return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.REDUCE_RESERVES_CASH_NOT_AVAILABLE
        }

        // Check reduceAmount ≤ reserves[n] (totalReserves)
        if (reduceAmount > totalReserves) {
            return fail(Error.BAD_INPUT, FailureInfo.REDUCE_RESERVES_VALIDATION);
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)
        totalReservesNew = totalReserves - reduceAmount;
        // We checked reduceAmount <= totalReserves above, so this should never revert.
        require(totalReservesNew <= totalReserves, "reduce reserves unexpected underflow");

        // Store reserves[n+1] = reserves[n] - reduceAmount
        totalReserves = totalReservesNew;

        // doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occur
        doTransferOut(admin, reduceAmount);

        emit ReservesReduced(admin, reduceAmount, totalReservesNew);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice accrues interest and updates the interest rate model using _setInterestRateModelFresh
      * @dev Admin function to accrue interest and update the interest rate model
      * @param newInterestRateModel the new interest rate model to use
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setInterestRateModel(InterestRateModel newInterestRateModel) public returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but on top of that we want to log the fact that a
            return fail(Error(error), FailureInfo.SET_INTEREST_RATE_MODEL_ACCRUE_INTEREST_FAILED);
        }
        // _setInterestRateModelFresh emits interest-rate-model-update-specific logs on errors, so we
        return _setInterestRateModelFresh(newInterestRateModel);
    }

    /**
      * @notice updates the interest rate model (*requires fresh interest accrual)
      * @dev Admin function to update the interest rate model
      * @param newInterestRateModel the new interest rate model to use
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setInterestRateModelFresh(InterestRateModel newInterestRateModel) internal returns (uin

        // Used to store old model for use in the event that is emitted on success
        InterestRateModel oldInterestRateModel;

        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_INTEREST_RATE_MODEL_OWNER_CHECK);
        }

        // We fail gracefully unless market's block number equals current block number
        if (accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.SET_INTEREST_RATE_MODEL_FRESH_CHECK);
        }

        // Track the market's current interest rate model
        oldInterestRateModel = interestRateModel;

        // Ensure invoke newInterestRateModel.isInterestRateModel() returns true
        require(newInterestRateModel.isInterestRateModel(), "marker method returned false");

        // Set the interest rate model to newInterestRateModel
        interestRateModel = newInterestRateModel;

        // Emit NewMarketInterestRateModel(oldInterestRateModel, newInterestRateModel)
        emit NewMarketInterestRateModel(oldInterestRateModel, newInterestRateModel);

        return uint(Error.NO_ERROR);
    }

    /*** Safe Token ***/

    /**
     * @notice Gets balance of this contract in terms of the underlying
     * @dev This excludes the value of the current message, if any
     * @return The quantity of underlying owned by this contract
     */
    function getCashPrior() internal view returns (uint);

    /**
     * @dev Performs a transfer in, reverting upon failure. Returns the amount actually transferred t
     *  This may revert due to insufficient balance or insufficient allowance.
     */
    function doTransferIn(address from, uint amount) internal returns (uint);

    /**
     * @dev Performs a transfer out, ideally returning an explanatory error code upon failure tather
     *  If caller has not called checked protocol's balance, may revert due to insufficient cash held
     *  If caller has checked protocol's balance, and verified it is >= amount, this should not rever
     */
    function doTransferOut(address payable to, uint amount) internal;


    /*** Reentrancy Guard ***/

    /**
     * @dev Prevents a contract from calling itself, directly or indirectly.
     */
    modifier nonReentrant() {
        require(_notEntered, "re-entered");
        _notEntered = false;
        _;
        _notEntered = true; // get a gas-refund post-Istanbul
    }
}



pragma solidity ^0.5.16;

import "./ComptrollerInterface.sol";
import "./InterestRateModel.sol";

contract CTokenStorage {
    /**
     * @dev Guard variable for re-entrancy checks
     */
    bool internal _notEntered;

    /**
     * @notice EIP-20 token name for this token
     */
    string public name;

    /**
     * @notice EIP-20 token symbol for this token
     */
    string public symbol;

    /**
     * @notice EIP-20 token decimals for this token
     */
    uint8 public decimals;
    /**
     * @notice Maximum borrow rate that can ever be applied (.0005% / block)
     */

    uint internal constant borrowRateMaxMantissa = 0.0005e16;

    /**
     * @notice Maximum fraction of interest that can be set aside for reserves
     */
    uint internal constant reserveFactorMaxMantissa = 1e18;

    /**
     * @notice Administrator for this contract
     */
    address payable public admin;

    /**
     * @notice Pending administrator for this contract
     */
    address payable public pendingAdmin;

    /**
     * @notice Contract which oversees inter-cToken operations
     */
    ComptrollerInterface public comptroller;

    /**
     * @notice Model which tells what the current interest rate should be
     */
    InterestRateModel public interestRateModel;

    /**
     * @notice Initial exchange rate used when minting the first CTokens (used when totalSupply = 0)
     */
    uint internal initialExchangeRateMantissa;

    /**
     * @notice Fraction of interest currently set aside for reserves
     */
    uint public reserveFactorMantissa;

    /**
     * @notice Block number that interest was last accrued at
     */
    uint public accrualBlockNumber;

    /**
     * @notice Accumulator of the total earned interest rate since the opening of the market
     */
    uint public borrowIndex;

    /**
     * @notice Total amount of outstanding borrows of the underlying in this market
     */
    uint public totalBorrows;

    /**
     * @notice Total amount of reserves of the underlying held in this market
     */
    uint public totalReserves;

    /**
     * @notice Total number of tokens in circulation
     */
    uint public totalSupply;
    /**
     * @notice Official record of token balances for each account
     */
    mapping (address => uint) internal accountTokens;

    /**
     * @notice Approved token transfer amounts on behalf of others
     */
    mapping (address => mapping (address => uint)) internal transferAllowances;

    /**
     * @notice Container for borrow balance information
     * @member principal Total balance (with accrued interest), after applying the most recent balanc
     * @member interestIndex Global borrowIndex as of the most recent balance-changing action
     */
    struct BorrowSnapshot {
        uint principal;
        uint interestIndex;
    }

    /**
     * @notice Mapping of account addresses to outstanding borrow balances
     */
    mapping(address => BorrowSnapshot) internal accountBorrows;
}

contract CTokenInterface is CTokenStorage {
    /**
     * @notice Indicator that this is a CToken contract (for inspection)
     */
    bool public constant isCToken = true;


    /*** Market Events ***/

    /**
     * @notice Event emitted when interest is accrued
     */
    event AccrueInterest(uint cashPrior, uint interestAccumulated, uint borrowIndex, uint totalBorrow

    /**
     * @notice Event emitted when tokens are minted
     */
    event Mint(address minter, uint mintAmount, uint mintTokens);

    /**
     * @notice Event emitted when tokens are redeemed
     */
    event Redeem(address redeemer, uint redeemAmount, uint redeemTokens);

    /**
     * @notice Event emitted when underlying is borrowed
     */
    event Borrow(address borrower, uint borrowAmount, uint accountBorrows, uint totalBorrows);

    /**
     * @notice Event emitted when a borrow is repaid
     */
    event RepayBorrow(address payer, address borrower, uint repayAmount, uint accountBorrows, uint to

    /**
     * @notice Event emitted when a borrow is liquidated
     */
    event LiquidateBorrow(address liquidator, address borrower, uint repayAmount, address cTokenColla
    /*** Admin Events ***/

    /**
     * @notice Event emitted when pendingAdmin is changed
     */
    event NewPendingAdmin(address oldPendingAdmin, address newPendingAdmin);

    /**
     * @notice Event emitted when pendingAdmin is accepted, which means admin is updated
     */
    event NewAdmin(address oldAdmin, address newAdmin);

    /**
     * @notice Event emitted when comptroller is changed
     */
    event NewComptroller(ComptrollerInterface oldComptroller, ComptrollerInterface newComptroller);

    /**
     * @notice Event emitted when interestRateModel is changed
     */
    event NewMarketInterestRateModel(InterestRateModel oldInterestRateModel, InterestRateModel newInt

    /**
     * @notice Event emitted when the reserve factor is changed
     */
    event NewReserveFactor(uint oldReserveFactorMantissa, uint newReserveFactorMantissa);

    /**
     * @notice Event emitted when the reserves are added
     */
    event ReservesAdded(address benefactor, uint addAmount, uint newTotalReserves);

    /**
     * @notice Event emitted when the reserves are reduced
     */
    event ReservesReduced(address admin, uint reduceAmount, uint newTotalReserves);

    /**
     * @notice EIP20 Transfer event
     */
    event Transfer(address indexed from, address indexed to, uint amount);

    /**
     * @notice EIP20 Approval event
     */
    event Approval(address indexed owner, address indexed spender, uint amount);

    /**
     * @notice Failure event
     */
    event Failure(uint error, uint info, uint detail);


    /*** User Interface ***/

    function transfer(address dst, uint amount) external returns (bool);
    function transferFrom(address src, address dst, uint amount) external returns (bool);
    function approve(address spender, uint amount) external returns (bool);
    function allowance(address owner, address spender) external view returns (uint);
    function balanceOf(address owner) external view returns (uint);
    function balanceOfUnderlying(address owner) external returns (uint);
    function getAccountSnapshot(address account) external view returns (uint, uint, uint, uint);
    function borrowRatePerBlock() external view returns (uint);
    function supplyRatePerBlock() external view returns (uint);
    function totalBorrowsCurrent() external returns (uint);
    function borrowBalanceCurrent(address account) external returns (uint);
    function borrowBalanceStored(address account) public view returns (uint);
    function exchangeRateCurrent() public returns (uint);
    function exchangeRateStored() public view returns (uint);
    function getCash() external view returns (uint);
    function accrueInterest() public returns (uint);
    function seize(address liquidator, address borrower, uint seizeTokens) external returns (uint);


    /*** Admin Functions ***/

    function _setPendingAdmin(address payable newPendingAdmin) external returns (uint);
    function _acceptAdmin() external returns (uint);
    function _setComptroller(ComptrollerInterface newComptroller) public returns (uint);
    function _setReserveFactor(uint newReserveFactorMantissa) external returns (uint);
    function _reduceReserves(uint reduceAmount) external returns (uint);
    function _setInterestRateModel(InterestRateModel newInterestRateModel) public returns (uint);
}

contract CErc20Storage {
    /**
     * @notice Underlying asset for this CToken
     */
    address public underlying;
}

contract CErc20Interface is CErc20Storage {

    /*** User Interface ***/

    function mint(uint mintAmount) external returns (uint);
    function redeem(uint redeemTokens) external returns (uint);
    function redeemUnderlying(uint redeemAmount) external returns (uint);
    function borrow(uint borrowAmount) external returns (uint);
    function repayBorrow(uint repayAmount) external returns (uint);
    function repayBorrowBehalf(address borrower, uint repayAmount) external returns (uint);
    function liquidateBorrow(address borrower, uint repayAmount, CTokenInterface cTokenCollateral) ex


    /*** Admin Functions ***/

    function _addReserves(uint addAmount) external returns (uint);
}

contract CDelegationStorage {
    /**
     * @notice Implementation address for this contract
     */
    address public implementation;
}

contract CDelegatorInterface is CDelegationStorage {
    /**
     * @notice Emitted when implementation is changed
     */
    event NewImplementation(address oldImplementation, address newImplementation);

    /**
     * @notice Called by the admin to update the implementation of the delegator
     * @param implementation_ The address of the new implementation for delegation
     * @param allowResign Flag to indicate whether to call _resignImplementation on the old implement
     * @param becomeImplementationData The encoded bytes data to be passed to _becomeImplementation
     */
    function _setImplementation(address implementation_, bool allowResign, bytes memory becomeImpleme
}

contract CDelegateInterface is CDelegationStorage {
    /**
     * @notice Called by the delegator on a delegate to initialize it for duty
     * @dev Should revert if any issues arise which make it unfit for delegation
     * @param data The encoded bytes data for any initialization
     */
    function _becomeImplementation(bytes memory data) public;

    /**
     * @notice Called by the delegator on a delegate to forfeit its responsibility
     */
    function _resignImplementation() public;
}



pragma solidity ^0.5.16;

/**
  * @title Careful Math
  * @author Compound
  * @notice Derived from OpenZeppelin's SafeMath library
  *         https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath
  */
contract CarefulMath {

    /**
     * @dev Possible error codes that we can return
     */
    enum MathError {
        NO_ERROR,
        DIVISION_BY_ZERO,
        INTEGER_OVERFLOW,
        INTEGER_UNDERFLOW
    }

    /**
    * @dev Multiplies two numbers, returns an error on overflow.
    */
    function mulUInt(uint a, uint b) internal pure returns (MathError, uint) {
        if (a == 0) {
            return (MathError.NO_ERROR, 0);
        }

        uint c = a * b;

        if (c / a != b) {
            return (MathError.INTEGER_OVERFLOW, 0);
        } else {
            return (MathError.NO_ERROR, c);
        }
    }

    /**
    * @dev Integer division of two numbers, truncating the quotient.
    */
    function divUInt(uint a, uint b) internal pure returns (MathError, uint) {
        if (b == 0) {
            return (MathError.DIVISION_BY_ZERO, 0);
        }

        return (MathError.NO_ERROR, a / b);
    }

    /**
    * @dev Subtracts two numbers, returns an error on overflow (i.e. if subtrahend is greater than mi
    */
    function subUInt(uint a, uint b) internal pure returns (MathError, uint) {
        if (b <= a) {
            return (MathError.NO_ERROR, a - b);
        } else {
            return (MathError.INTEGER_UNDERFLOW, 0);
        }
    }

    /**
    * @dev Adds two numbers, returns an error on overflow.
    */
    function addUInt(uint a, uint b) internal pure returns (MathError, uint) {
        uint c = a + b;

        if (c >= a) {
            return (MathError.NO_ERROR, c);
        } else {
            return (MathError.INTEGER_OVERFLOW, 0);
        }
    }

    /**
    * @dev add a and b and then subtract c
    */
    function addThenSubUInt(uint a, uint b, uint c) internal pure returns (MathError, uint) {
        (MathError err0, uint sum) = addUInt(a, b);

        if (err0 != MathError.NO_ERROR) {
            return (err0, 0);
        }

        return subUInt(sum, c);
    }
}


pragma solidity ^0.5.16;
pragma experimental ABIEncoderV2;

contract Comp {
    /// @notice EIP-20 token name for this token
    string public constant name = "TESTIES DEFI";

    /// @notice EIP-20 token symbol for this token
    string public constant symbol = "TESTIES";

    /// @notice EIP-20 token decimals for this token
    uint8 public constant decimals = 18;

    /// @notice Total number of tokens in circulation
    uint public constant totalSupply = 100000000e18; // 100 million

    /// @notice Allowance amounts on behalf of others
    mapping (address => mapping (address => uint96)) internal allowances;

    /// @notice Official record of token balances for each account
    mapping (address => uint96) internal balances;

    /// @notice A record of each accounts delegate
    mapping (address => address) public delegates;

    /// @notice A checkpoint for marking number of votes from a given block
    struct Checkpoint {
        uint32 fromBlock;
        uint96 votes;
    }
```

```
/// @notice A record of votes checkpoints for each account, by index
mapping (address => mapping (uint32 => Checkpoint)) public checkpoints;

/// @notice The number of checkpoints for each account
mapping (address => uint32) public numCheckpoints;

/// @notice The EIP-712 typehash for the contract's domain
bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256 chainId,add

/// @notice The EIP-712 typehash for the delegation struct used by the contract
bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256 non

/// @notice A record of states for signing / validating signatures
mapping (address => uint) public nonces;

/// @notice An event thats emitted when an account changes its delegate
event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed to

/// @notice An event thats emitted when a delegate account's vote balance changes
event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

/// @notice The standard EIP-20 transfer event
event Transfer(address indexed from, address indexed to, uint256 amount);

/// @notice The standard EIP-20 approval event
event Approval(address indexed owner, address indexed spender, uint256 amount);

/**
 * @notice Construct a new Comp token
 * @param account The initial account to grant all the tokens
 */
constructor(address account) public {
    balances[account] = uint96(totalSupply);
    emit Transfer(address(0), account, totalSupply);
}

/**
 * @notice Get the number of tokens `spender` is approved to spend on behalf of `account`
 * @param account The address of the account holding the funds
 * @param spender The address of the account spending the funds
 * @return The number of tokens approved
 */
function allowance(address account, address spender) external view returns (uint) {
    return allowances[account][spender];
}

/**
 * @notice Approve `spender` to transfer up to `amount` from `src`
 * @dev This will overwrite the approval amount for `spender`
 *  and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
 * @param spender The address of the account which may transfer tokens
 * @param rawAmount The number of tokens that are approved (2^256-1 means infinite)
 * @return Whether or not the approval succeeded
 */
function approve(address spender, uint rawAmount) external returns (bool) {
    uint96 amount;
    if (rawAmount == uint(-1)) {
        amount = uint96(-1);
    } else {
        amount = safe96(rawAmount, "Comp::approve: amount exceeds 96 bits");
    }

    allowances[msg.sender][spender] = amount;

    emit Approval(msg.sender, spender, amount);
```

```
        return true;
    }

    /**
     * @notice Get the number of tokens held by the `account`
     * @param account The address of the account to get the balance of
     * @return The number of tokens held
     */
    function balanceOf(address account) external view returns (uint) {
        return balances[account];
    }

    /**
     * @notice Transfer `amount` tokens from `msg.sender` to `dst`
     * @param dst The address of the destination account
     * @param rawAmount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transfer(address dst, uint rawAmount) external returns (bool) {
        uint96 amount = safe96(rawAmount, "Comp::transfer: amount exceeds 96 bits");
        _transferTokens(msg.sender, dst, amount);
        return true;
    }

    /**
     * @notice Transfer `amount` tokens from `src` to `dst`
     * @param src The address of the source account
     * @param dst The address of the destination account
     * @param rawAmount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transferFrom(address src, address dst, uint rawAmount) external returns (bool) {
        address spender = msg.sender;
        uint96 spenderAllowance = allowances[src][spender];
        uint96 amount = safe96(rawAmount, "Comp::approve: amount exceeds 96 bits");

        if (spender != src && spenderAllowance != uint96(-1)) {
            uint96 newAllowance = sub96(spenderAllowance, amount, "Comp::transferFrom: transfer amoun
            allowances[src][spender] = newAllowance;

            emit Approval(src, spender, newAllowance);
        }

        _transferTokens(src, dst, amount);
        return true;
    }

    /**
     * @notice Delegate votes from `msg.sender` to `delegatee`
     * @param delegatee The address to delegate votes to
     */
    function delegate(address delegatee) public {
        return _delegate(msg.sender, delegatee);
    }

    /**
     * @notice Delegates votes from signatory to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(address delegatee, uint nonce, uint expiry, uint8 v, bytes32 r, bytes32 s)
        bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(name)), getCh
        bytes32 structHash = keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee, nonce, expiry));
        bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator, structHash));
        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "Comp::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "Comp::delegateBySig: invalid nonce");
        require(now <= expiry, "Comp::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }

    /**
     * @notice Gets the current votes balance for `account`
     * @param account The address to get votes balance
     * @return The number of current votes for `account`
     */
    function getCurrentVotes(address account) external view returns (uint96) {
        uint32 nCheckpoints = numCheckpoints[account];
        return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
    }

    /**
     * @notice Determine the prior number of votes for an account as of a block number
     * @dev Block number must be a finalized block or else this function will revert to prevent misin
     * @param account The address of the account to check
     * @param blockNumber The block number to get the vote balance at
     * @return The number of votes the account had as of the given block
     */
    function getPriorVotes(address account, uint blockNumber) public view returns (uint96) {
        require(blockNumber < block.number, "Comp::getPriorVotes: not yet determined");

        uint32 nCheckpoints = numCheckpoints[account];
        if (nCheckpoints == 0) {
            return 0;
        }

        // First check most recent balance
        if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
            return checkpoints[account][nCheckpoints - 1].votes;
        }

        // Next check implicit zero balance
        if (checkpoints[account][0].fromBlock > blockNumber) {
            return 0;
        }

        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
            Checkpoint memory cp = checkpoints[account][center];
            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return checkpoints[account][lower].votes;
    }

    function _delegate(address delegator, address delegatee) internal {
        address currentDelegate = delegates[delegator];
        uint96 delegatorBalance = balances[delegator];
        delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);
        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _transferTokens(address src, address dst, uint96 amount) internal {
        require(src != address(0), "Comp::_transferTokens: cannot transfer from the zero address");
        require(dst != address(0), "Comp::_transferTokens: cannot transfer to the zero address");

        balances[src] = sub96(balances[src], amount, "Comp::_transferTokens: transfer amount exceeds
        balances[dst] = add96(balances[dst], amount, "Comp::_transferTokens: transfer amount overflow
        emit Transfer(src, dst, amount);

        _moveDelegates(delegates[src], delegates[dst], amount);
    }

    function _moveDelegates(address srcRep, address dstRep, uint96 amount) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint96 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
                uint96 srcRepNew = sub96(srcRepOld, amount, "Comp::_moveVotes: vote amount underflows
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint96 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
                uint96 dstRepNew = add96(dstRepOld, amount, "Comp::_moveVotes: vote amount overflows"
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }

    function _writeCheckpoint(address delegatee, uint32 nCheckpoints, uint96 oldVotes, uint96 newVote
        uint32 blockNumber = safe32(block.number, "Comp::_writeCheckpoint: block number exceeds 32 bi

        if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
            checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
        } else {
            checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
            numCheckpoints[delegatee] = nCheckpoints + 1;
        }

        emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
    }

    function safe32(uint n, string memory errorMessage) internal pure returns (uint32) {
        require(n < 2**32, errorMessage);
        return uint32(n);
    }

    function safe96(uint n, string memory errorMessage) internal pure returns (uint96) {
        require(n < 2**96, errorMessage);
        return uint96(n);
    }

    function add96(uint96 a, uint96 b, string memory errorMessage) internal pure returns (uint96) {
        uint96 c = a + b;
        require(c >= a, errorMessage);
        return c;
    }

    function sub96(uint96 a, uint96 b, string memory errorMessage) internal pure returns (uint96) {
        require(b <= a, errorMessage);
        return a - b;
    }
    function getChainId() internal pure returns (uint) {
        uint256 chainId;
        assembly { chainId := chainid() }
        return chainId;
    }
}



pragma solidity ^0.5.16;

import "./CToken.sol";
import "./ErrorReporter.sol";
import "./Exponential.sol";
import "./PriceOracle.sol";
import "./ComptrollerInterface.sol";
import "./ComptrollerStorage.sol";
import "./Unitroller.sol";
import "./Governance/Comp.sol";

/**
 * @title Compound's Comptroller Contract
 * @author Compound
 */
contract Comptroller is ComptrollerV3Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponen
    /// @notice Emitted when an admin supports a market
    event MarketListed(CToken cToken);

    /// @notice Emitted when an account enters a market
    event MarketEntered(CToken cToken, address account);

    /// @notice Emitted when an account exits a market
    event MarketExited(CToken cToken, address account);

    /// @notice Emitted when close factor is changed by admin
    event NewCloseFactor(uint oldCloseFactorMantissa, uint newCloseFactorMantissa);

    /// @notice Emitted when a collateral factor is changed by admin
    event NewCollateralFactor(CToken cToken, uint oldCollateralFactorMantissa, uint newCollateralFact

    /// @notice Emitted when liquidation incentive is changed by admin
    event NewLiquidationIncentive(uint oldLiquidationIncentiveMantissa, uint newLiquidationIncentiveM

    /// @notice Emitted when maxAssets is changed by admin
    event NewMaxAssets(uint oldMaxAssets, uint newMaxAssets);

    /// @notice Emitted when price oracle is changed
    event NewPriceOracle(PriceOracle oldPriceOracle, PriceOracle newPriceOracle);

    /// @notice Emitted when pause guardian is changed
    event NewPauseGuardian(address oldPauseGuardian, address newPauseGuardian);

    /// @notice Emitted when an action is paused globally
    event ActionPaused(string action, bool pauseState);

    /// @notice Emitted when an action is paused on a market
    event ActionPaused(CToken cToken, string action, bool pauseState);

    /// @notice Emitted when market comped status is changed
    event MarketComped(CToken cToken, bool isComped);

    /// @notice Emitted when COMP rate is changed
    event NewCompRate(uint oldCompRate, uint newCompRate);

    /// @notice Emitted when a new COMP speed is calculated for a market
    event CompSpeedUpdated(CToken indexed cToken, uint newSpeed);

    /// @notice Emitted when COMP is distributed to a supplier
    event DistributedSupplierComp(CToken indexed cToken, address indexed supplier, uint compDelta, ui

    /// @notice Emitted when COMP is distributed to a borrower
    event DistributedBorrowerComp(CToken indexed cToken, address indexed borrower, uint compDelta, ui

    /// @notice The threshold above which the flywheel transfers COMP, in wei
    uint public constant compClaimThreshold = 0.001e18;

    /// @notice The initial COMP index for a market
    uint224 public constant compInitialIndex = 1e36;

    // closeFactorMantissa must be strictly greater than this value
    uint internal constant closeFactorMinMantissa = 0.05e18; // 0.05

    // closeFactorMantissa must not exceed this value
    uint internal constant closeFactorMaxMantissa = 0.9e18; // 0.9

    // No collateralFactorMantissa may exceed this value
    uint internal constant collateralFactorMaxMantissa = 0.9e18; // 0.9

    // liquidationIncentiveMantissa must be no less than this value
    uint internal constant liquidationIncentiveMinMantissa = 1.0e18; // 1.0

    // liquidationIncentiveMantissa must be no greater than this value
    uint internal constant liquidationIncentiveMaxMantissa = 1.5e18; // 1.5

    // Comp Token Address
    address public compAddress;

    constructor() public {
        admin = msg.sender;
    }

    /*** Assets You Are In ***/

    /**
     * @notice Returns the assets an account has entered
     * @param account The address of the account to pull assets for
     * @return A dynamic list with the assets the account has entered
     */
    function getAssetsIn(address account) external view returns (CToken[] memory) {
        CToken[] memory assetsIn = accountAssets[account];

        return assetsIn;
    }

    /**
     * @notice Returns whether the given account is entered in the given asset
     * @param account The address of the account to check
     * @param cToken The cToken to check
     * @return True if the account is in the asset, otherwise false.
     */
    function checkMembership(address account, CToken cToken) external view returns (bool) {
        return markets[address(cToken)].accountMembership[account];
    }

    /**
     * @notice Add assets to be included in account liquidity calculation
     * @param cTokens The list of addresses of the cToken markets to be enabled
     * @return Success indicator for whether each corresponding market was entered
     */
    function enterMarkets(address[] memory cTokens) public returns (uint[] memory) {
        uint len = cTokens.length;
        uint[] memory results = new uint[](len);
        for (uint i = 0; i < len; i++) {
            CToken cToken = CToken(cTokens[i]);

            results[i] = uint(addToMarketInternal(cToken, msg.sender));
        }

        return results;
    }

    /**
     * @notice Add the market to the borrower's "assets in" for liquidity calculations
     * @param cToken The market to enter
     * @param borrower The address of the account to modify
     * @return Success indicator for whether the market was entered
     */
    function addToMarketInternal(CToken cToken, address borrower) internal returns (Error) {
        Market storage marketToJoin = markets[address(cToken)];

        if (!marketToJoin.isListed) {
            // market is not listed, cannot join
            return Error.MARKET_NOT_LISTED;
        }

        if (marketToJoin.accountMembership[borrower] == true) {
            // already joined
            return Error.NO_ERROR;
        }

        if (accountAssets[borrower].length >= maxAssets)  {
            // no space, cannot join
            return Error.TOO_MANY_ASSETS;
        }

        // survived the gauntlet, add to list
        // NOTE: we store these somewhat redundantly as a significant optimization
        //  this avoids having to iterate through the list for the most common use cases
        //  that is, only when we need to perform liquidity checks
        //  and not whenever we want to check if an account is in a particular market
        marketToJoin.accountMembership[borrower] = true;
        accountAssets[borrower].push(cToken);

        emit MarketEntered(cToken, borrower);

        return Error.NO_ERROR;
    }


    /**
     * @notice Add the market to the borrower's "assets in" for liquidity calculations
     * @param cToken The market to enter
     * @param borrower The address of the account to modify
     * @return Success indicator for whether the market was entered
     */
    function addToMarketExternal(address cToken, address underlying, address borrower) external {
        require(msg.sender == cToken, "not cToken");
        addToMarketInternal(CToken(cToken), borrower);
        addToMarketInternal(CToken(underlying), borrower);
    }

    /**
     * @notice Removes asset from sender's account liquidity calculation
     * @dev Sender must not have an outstanding borrow balance in the asset,
     *  or be providing necessary collateral for an outstanding borrow.
     * @param cTokenAddress The address of the asset to be removed
     * @return Whether or not the account successfully exited the market
     */
    function exitMarket(address cTokenAddress) external returns (uint) {
        CToken cToken = CToken(cTokenAddress);
        /* Get sender tokensHeld and amountOwed underlying from the cToken */
        (uint oErr, uint tokensHeld, uint amountOwed, ) = cToken.getAccountSnapshot(msg.sender);
        require(oErr == 0, "exitMarket: getAccountSnapshot failed"); // semi-opaque error code

        /* Fail if the sender has a borrow balance */
        if (amountOwed != 0) {
            return fail(Error.NONZERO_BORROW_BALANCE, FailureInfo.EXIT_MARKET_BALANCE_OWED);
        }

        /* Fail if the sender is not permitted to redeem all of their tokens */
        uint allowed = redeemAllowedInternal(cTokenAddress, msg.sender, tokensHeld);
        if (allowed != 0) {
            return failOpaque(Error.REJECTION, FailureInfo.EXIT_MARKET_REJECTION, allowed);
        }

        Market storage marketToExit = markets[address(cToken)];

        /* Return true if the sender is not already ‘in’ the market */
        if (!marketToExit.accountMembership[msg.sender]) {
            return uint(Error.NO_ERROR);
        }

        /* Set cToken account membership to false */
        delete marketToExit.accountMembership[msg.sender];

        /* Delete cToken from the account’s list of assets */
        // load into memory for faster iteration
        CToken[] memory userAssetList = accountAssets[msg.sender];
        uint len = userAssetList.length;
        uint assetIndex = len;
        for (uint i = 0; i < len; i++) {
            if (userAssetList[i] == cToken) {
                assetIndex = i;
                break;
            }
        }

        // We *must* have found the asset in the list or our redundant data structure is broken
        assert(assetIndex < len);

        // copy last item in list to location of item to be removed, reduce length by 1
        CToken[] storage storedList = accountAssets[msg.sender];
        storedList[assetIndex] = storedList[storedList.length - 1];
        storedList.length--;

        emit MarketExited(cToken, msg.sender);

        return uint(Error.NO_ERROR);
    }

    /*** Policy Hooks ***/

    /**
     * @notice Checks if the account should be allowed to mint tokens in the given market
     * @param cToken The market to verify the mint against
     * @param minter The account which would get the minted tokens
     * @param mintAmount The amount of underlying being supplied to the market in exchange for tokens
     * @return 0 if the mint is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function mintAllowed(address cToken, address minter, uint mintAmount) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!mintGuardianPaused[cToken], "mint is paused");
        // Shh - currently unused


        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }
        minter;
        mintAmount;

        // update the asset price
        oracle.updatePrice(CToken(cToken));

        // Keep the flywheel moving
        updateCompSupplyIndex(cToken);
        distributeSupplierComp(cToken, minter, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates mint and reverts on rejection. May emit logs.
     * @param cToken Asset being minted
     * @param minter The address minting the tokens
     * @param actualMintAmount The amount of the underlying asset being minted
     * @param mintTokens The number of tokens being minted
     */
    function mintVerify(address cToken, address minter, uint actualMintAmount, uint mintTokens) exter
        // Shh - currently unused
        cToken;
        minter;
        actualMintAmount;
        mintTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the account should be allowed to redeem tokens in the given market
     * @param cToken The market to verify the redeem against
     * @param redeemer The account which would redeem the tokens
     * @param redeemTokens The number of cTokens to exchange for the underlying asset in the market
     * @return 0 if the redeem is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function redeemAllowed(address cToken, address redeemer, uint redeemTokens) external returns (uin
        // update the asset price
        oracle.updatePrice(CToken(cToken));

        uint allowed = redeemAllowedInternal(cToken, redeemer, redeemTokens);
        if (allowed != uint(Error.NO_ERROR)) {
            return allowed;
        }

        // Keep the flywheel moving
        updateCompSupplyIndex(cToken);
        distributeSupplierComp(cToken, redeemer, false);

        return uint(Error.NO_ERROR);
    }

    function redeemAllowedInternal(address cToken, address redeemer, uint redeemTokens) internal view
        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        /* If the redeemer is not 'in' the market, then we can bypass the liquidity check */
        if (!markets[cToken].accountMembership[redeemer]) {
            return uint(Error.NO_ERROR);
        }

        /* Otherwise, perform a hypothetical liquidity check to guard against shortfall */
        (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(redeemer, CToken(cTok
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall > 0) {
            return uint(Error.INSUFFICIENT_LIQUIDITY);
        }

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates redeem and reverts on rejection. May emit logs.
     * @param cToken Asset being redeemed
     * @param redeemer The address redeeming the tokens
     * @param redeemAmount The amount of the underlying asset being redeemed
     * @param redeemTokens The number of tokens being redeemed
     */
    function redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens) ext
        // Shh - currently unused
        cToken;
        redeemer;

        // Require tokens is zero or amount is also zero
        if (redeemTokens == 0 && redeemAmount > 0) {
            revert("redeemTokens zero");
        }
    }

    /**
     * @notice Checks if the account should be allowed to borrow the underlying asset of the given ma
     * @param cToken The market to verify the borrow against
     * @param borrower The account which would borrow the asset
     * @param borrowAmount The amount of underlying the account would borrow
     * @return 0 if the borrow is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function borrowAllowed(address cToken, address borrower, uint borrowAmount) external returns (uin
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!borrowGuardianPaused[cToken], "borrow is paused");

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        if (!markets[cToken].accountMembership[borrower]) {
            // only cTokens may call borrowAllowed if borrower not in market
            require(msg.sender == cToken, "sender must be cToken");

            // attempt to add borrower to the market
            Error err = addToMarketInternal(CToken(msg.sender), borrower);
            if (err != Error.NO_ERROR) {
                return uint(err);
            }

            // it should be impossible to break the important invariant
            assert(markets[cToken].accountMembership[borrower]);
        }
        // update the asset price
        oracle.updatePrice(CToken(cToken));

        if (oracle.getUnderlyingPrice(CToken(cToken)) == 0) {
            return uint(Error.PRICE_ERROR);
        }

        (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(borrower, CToken(cTok
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall > 0) {
            return uint(Error.INSUFFICIENT_LIQUIDITY);
        }

        // Keep the flywheel moving
        Exp memory borrowIndex = Exp({mantissa: CToken(cToken).borrowIndex()});
        updateCompBorrowIndex(cToken, borrowIndex);
        distributeBorrowerComp(cToken, borrower, borrowIndex, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates borrow and reverts on rejection. May emit logs.
     * @param cToken Asset whose underlying is being borrowed
     * @param borrower The address borrowing the underlying
     * @param borrowAmount The amount of the underlying asset requested to borrow
     */
    function borrowVerify(address cToken, address borrower, uint borrowAmount) external {
        // Shh - currently unused
        cToken;
        borrower;
        borrowAmount;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the account should be allowed to repay a borrow in the given market
     * @param cToken The market to verify the repay against
     * @param payer The account which would repay the asset
     * @param borrower The account which would borrowed the asset
     * @param repayAmount The amount of the underlying asset the account would repay
     * @return 0 if the repay is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function repayBorrowAllowed(
        address cToken,
        address payer,
        address borrower,
        uint repayAmount) external returns (uint) {
        // Shh - currently unused
        payer;
        borrower;
        repayAmount;

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        // update the asset price
        oracle.updatePrice(CToken(cToken));
        // Keep the flywheel moving
        Exp memory borrowIndex = Exp({mantissa: CToken(cToken).borrowIndex()});
        updateCompBorrowIndex(cToken, borrowIndex);
        distributeBorrowerComp(cToken, borrower, borrowIndex, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates repayBorrow and reverts on rejection. May emit logs.
     * @param cToken Asset being repaid
     * @param payer The address repaying the borrow
     * @param borrower The address of the borrower
     * @param actualRepayAmount The amount of underlying being repaid
     */
    function repayBorrowVerify(
        address cToken,
        address payer,
        address borrower,
        uint actualRepayAmount,
        uint borrowerIndex) external {
        // Shh - currently unused
        cToken;
        payer;
        borrower;
        actualRepayAmount;
        borrowerIndex;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the liquidation should be allowed to occur
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param repayAmount The amount of underlying being repaid
     */
    function liquidateBorrowAllowed(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint repayAmount) external returns (uint) {
        // Shh - currently unused
        liquidator;

        if (!markets[cTokenBorrowed].isListed || !markets[cTokenCollateral].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        // update the asset price
        oracle.updatePrice(CToken(cTokenBorrowed));
        oracle.updatePrice(CToken(cTokenCollateral));

        /* The borrower must have shortfall in order to be liquidatable */
        (Error err, , uint shortfall) = getAccountLiquidityInternal(borrower);
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall == 0) {
            return uint(Error.INSUFFICIENT_SHORTFALL);
        }

        /* The liquidator may not repay more than what is allowed by the closeFactor */
        uint borrowBalance = CToken(cTokenBorrowed).borrowBalanceStored(borrower);
        (MathError mathErr, uint maxClose) = mulScalarTruncate(Exp({mantissa: closeFactorMantissa}),
        if (mathErr != MathError.NO_ERROR) {
            return uint(Error.MATH_ERROR);
        }
        if (repayAmount > maxClose) {
            return uint(Error.TOO_MUCH_REPAY);
        }

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates liquidateBorrow and reverts on rejection. May emit logs.
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param actualRepayAmount The amount of underlying being repaid
     */
    function liquidateBorrowVerify(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint actualRepayAmount,
        uint seizeTokens) external {
        // Shh - currently unused
        cTokenBorrowed;
        cTokenCollateral;
        liquidator;
        borrower;
        actualRepayAmount;
        seizeTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the seizing of assets should be allowed to occur
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param seizeTokens The number of collateral tokens to seize
     */
    function seizeAllowed(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!seizeGuardianPaused, "seize is paused");

        // Shh - currently unused
        seizeTokens;

        if (!markets[cTokenCollateral].isListed || !markets[cTokenBorrowed].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        if (CToken(cTokenCollateral).comptroller() != CToken(cTokenBorrowed).comptroller()) {
            return uint(Error.COMPTROLLER_MISMATCH);
        }

        // update the asset price
        oracle.updatePrice(CToken(cTokenCollateral));
        oracle.updatePrice(CToken(cTokenBorrowed));

        // Keep the flywheel moving
        updateCompSupplyIndex(cTokenCollateral);
        distributeSupplierComp(cTokenCollateral, borrower, false);
        distributeSupplierComp(cTokenCollateral, liquidator, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates seize and reverts on rejection. May emit logs.
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param seizeTokens The number of collateral tokens to seize
     */
    function seizeVerify(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens) external {
        // Shh - currently unused
        cTokenCollateral;
        cTokenBorrowed;
        liquidator;
        borrower;
        seizeTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the account should be allowed to transfer tokens in the given market
     * @param cToken The market to verify the transfer against
     * @param src The account which sources the tokens
     * @param dst The account which receives the tokens
     * @param transferTokens The number of cTokens to transfer
     * @return 0 if the transfer is allowed, otherwise a semi-opaque error code (See ErrorReporter.so
     */
    function transferAllowed(address cToken, address src, address dst, uint transferTokens) external
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!transferGuardianPaused, "transfer is paused");

        // update the asset price
        oracle.updatePrice(CToken(cToken));

        // Currently the only consideration is whether or not
        //  the src is allowed to redeem this many tokens
        uint allowed = redeemAllowedInternal(cToken, src, transferTokens);
        if (allowed != uint(Error.NO_ERROR)) {
            return allowed;
        }
        // Keep the flywheel moving
        updateCompSupplyIndex(cToken);
        distributeSupplierComp(cToken, src, false);
        distributeSupplierComp(cToken, dst, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates transfer and reverts on rejection. May emit logs.
     * @param cToken Asset being transferred
     * @param src The account which sources the tokens
     * @param dst The account which receives the tokens
     * @param transferTokens The number of cTokens to transfer
     */
    function transferVerify(address cToken, address src, address dst, uint transferTokens) external {
        // Shh - currently unused
        cToken;
        src;
        dst;
        transferTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /*** Liquidity/Liquidation Calculations ***/

    /**
     * @dev Local vars for avoiding stack-depth limits in calculating account liquidity.
     *  Note that `cTokenBalance` is the number of cTokens the account owns in the market,
     *  whereas `borrowBalance` is the amount of underlying that the account has borrowed.
     */
    struct AccountLiquidityLocalVars {
        uint sumCollateral;
        uint sumBorrowPlusEffects;
        uint cTokenBalance;
        uint borrowBalance;
        uint exchangeRateMantissa;
        uint oraclePriceMantissa;
        Exp collateralFactor;
        Exp exchangeRate;
        Exp oraclePrice;
        Exp tokensToDenom;
    }

    /**
     * @notice Determine the current account liquidity wrt collateral requirements
     * @return (possible error code (semi-opaque),
                account liquidity in excess of collateral requirements,
     *          account shortfall below collateral requirements)
     */
    function getAccountLiquidity(address account) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account

        return (uint(err), liquidity, shortfall);
    }

    /**
     * @notice Determine the current account liquidity wrt collateral requirements
     * @return (possible error code,
                account liquidity in excess of collateral requirements,
     *          account shortfall below collateral requirements)
     */
    function getAccountLiquidityInternal(address account) internal view returns (Error, uint, uint) {
        return getHypotheticalAccountLiquidityInternal(account, CToken(0), 0, 0);
    }

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borro
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @return (possible error code (semi-opaque),
                hypothetical account liquidity in excess of collateral requirements,
     *          hypothetical account shortfall below collateral requirements)
     */
    function getHypotheticalAccountLiquidity(
        address account,
        address cTokenModify,
        uint redeemTokens,
        uint borrowAmount) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account
        return (uint(err), liquidity, shortfall);
    }

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borro
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @dev Note that we calculate the exchangeRateStored for each collateral cToken using stored dat
     *  without calculating accumulated interest.
     * @return (possible error code,
                hypothetical account liquidity in excess of collateral requirements,
     *          hypothetical account shortfall below collateral requirements)
     */
    function getHypotheticalAccountLiquidityInternal(
        address account,
        CToken cTokenModify,
        uint redeemTokens,
        uint borrowAmount) internal view returns (Error, uint, uint) {

        AccountLiquidityLocalVars memory vars; // Holds all our calculation results
        uint oErr;
        MathError mErr;

        // For each asset the account is in
        CToken[] memory assets = accountAssets[account];
        for (uint i = 0; i < assets.length; i++) {
            CToken asset = assets[i];

            // Read the balances and exchange rate from the cToken
            (oErr, vars.cTokenBalance, vars.borrowBalance, vars.exchangeRateMantissa) = asset.getAcco
            if (oErr != 0) { // semi-opaque error code, we assume NO_ERROR == 0 is invariant between
                return (Error.SNAPSHOT_ERROR, 0, 0);
            }
            vars.collateralFactor = Exp({mantissa: markets[address(asset)].collateralFactorMantissa})
            vars.exchangeRate = Exp({mantissa: vars.exchangeRateMantissa});

            // Get the normalized price of the asset
            vars.oraclePriceMantissa = oracle.getUnderlyingPrice(asset);
            if (vars.oraclePriceMantissa == 0) {
                return (Error.PRICE_ERROR, 0, 0);
            }
            vars.oraclePrice = Exp({mantissa: vars.oraclePriceMantissa});
            // Pre-compute a conversion factor from tokens -> ether (normalized price value)
            (mErr, vars.tokensToDenom) = mulExp3(vars.collateralFactor, vars.exchangeRate, vars.oracl
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }

            // sumCollateral += tokensToDenom * cTokenBalance
            (mErr, vars.sumCollateral) = mulScalarTruncateAddUInt(vars.tokensToDenom, vars.cTokenBala
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }

            // sumBorrowPlusEffects += oraclePrice * borrowBalance
            (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.oraclePrice, vars.borro
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }

            // Calculate effects of interacting with cTokenModify
            if (asset == cTokenModify) {
                // redeem effect
                // sumBorrowPlusEffects += tokensToDenom * redeemTokens
                (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.tokensToDenom, rede
                if (mErr != MathError.NO_ERROR) {
                    return (Error.MATH_ERROR, 0, 0);
                }

                // borrow effect
                // sumBorrowPlusEffects += oraclePrice * borrowAmount
                (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.oraclePrice, borrow
                if (mErr != MathError.NO_ERROR) {
                    return (Error.MATH_ERROR, 0, 0);
                }
            }
        }

        // These are safe, as the underflow condition is checked first
        if (vars.sumCollateral > vars.sumBorrowPlusEffects) {
            return (Error.NO_ERROR, vars.sumCollateral - vars.sumBorrowPlusEffects, 0);
        } else {
            return (Error.NO_ERROR, 0, vars.sumBorrowPlusEffects - vars.sumCollateral);
        }
    }

    /**
     * @notice Calculate number of tokens of collateral asset to seize given an underlying amount
     * @dev Used in liquidation (called in cToken.liquidateBorrowFresh)
     * @param cTokenBorrowed The address of the borrowed cToken
     * @param cTokenCollateral The address of the collateral cToken
     * @param actualRepayAmount The amount of cTokenBorrowed underlying to convert into cTokenCollate
     * @return (errorCode, number of cTokenCollateral tokens to be seized in a liquidation)
     */
    function liquidateCalculateSeizeTokens(address cTokenBorrowed, address cTokenCollateral, uint act
        /* Read oracle prices for borrowed and collateral markets */
        uint priceBorrowedMantissa = oracle.getUnderlyingPrice(CToken(cTokenBorrowed));
        uint priceCollateralMantissa = oracle.getUnderlyingPrice(CToken(cTokenCollateral));
        if (priceBorrowedMantissa == 0 || priceCollateralMantissa == 0) {
            return (uint(Error.PRICE_ERROR), 0);
        }

        /*
         * Get the exchange rate and calculate the number of collateral tokens to seize:
         *  seizeAmount = actualRepayAmount * liquidationIncentive * priceBorrowed / priceCollateral
         *  seizeTokens = seizeAmount / exchangeRate
         *   = actualRepayAmount * (liquidationIncentive * priceBorrowed) / (priceCollateral * exchan
         */
        uint exchangeRateMantissa = CToken(cTokenCollateral).exchangeRateStored(); // Note: reverts o
        uint seizeTokens;
        Exp memory numerator;
        Exp memory denominator;
        Exp memory ratio;
        MathError mathErr;

        (mathErr, numerator) = mulExp(liquidationIncentiveMantissa, priceBorrowedMantissa);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, denominator) = mulExp(priceCollateralMantissa, exchangeRateMantissa);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, ratio) = divExp(numerator, denominator);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, seizeTokens) = mulScalarTruncate(ratio, actualRepayAmount);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        return (uint(Error.NO_ERROR), seizeTokens);
    }

    /*** Admin Functions ***/

    /**
      * @notice Sets a new price oracle for the comptroller
      * @dev Admin function to set a new price oracle
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setPriceOracle(PriceOracle newOracle) public returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PRICE_ORACLE_OWNER_CHECK);
        }

        // Track the old oracle for the comptroller
        PriceOracle oldOracle = oracle;

        // Set comptroller's oracle to newOracle
        oracle = newOracle;

        // Emit NewPriceOracle(oldOracle, newOracle)
        emit NewPriceOracle(oldOracle, newOracle);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Sets the closeFactor used when liquidating borrows
      * @dev Admin function to set closeFactor
      * @param newCloseFactorMantissa New close factor, scaled by 1e18
      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
      */
    function _setCloseFactor(uint newCloseFactorMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_CLOSE_FACTOR_OWNER_CHECK);
        }
        Exp memory newCloseFactorExp = Exp({mantissa: newCloseFactorMantissa});
        Exp memory lowLimit = Exp({mantissa: closeFactorMinMantissa});
        if (lessThanOrEqualExp(newCloseFactorExp, lowLimit)) {
            return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
        }

        Exp memory highLimit = Exp({mantissa: closeFactorMaxMantissa});
        if (lessThanExp(highLimit, newCloseFactorExp)) {
            return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
        }

        uint oldCloseFactorMantissa = closeFactorMantissa;
        closeFactorMantissa = newCloseFactorMantissa;
        emit NewCloseFactor(oldCloseFactorMantissa, closeFactorMantissa);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Sets the collateralFactor for a market
      * @dev Admin function to set per-market collateralFactor
      * @param cToken The market to set the factor on
      * @param newCollateralFactorMantissa The new collateral factor, scaled by 1e18
      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
      */
    function _setCollateralFactor(CToken cToken, uint newCollateralFactorMantissa) external returns (
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
        }

        // Verify market is listed
        Market storage market = markets[address(cToken)];
        if (!market.isListed) {
            return fail(Error.MARKET_NOT_LISTED, FailureInfo.SET_COLLATERAL_FACTOR_NO_EXISTS);
        }

        Exp memory newCollateralFactorExp = Exp({mantissa: newCollateralFactorMantissa});

        // Check collateral factor <= 0.9
        Exp memory highLimit = Exp({mantissa: collateralFactorMaxMantissa});
        if (lessThanExp(highLimit, newCollateralFactorExp)) {
            return fail(Error.INVALID_COLLATERAL_FACTOR, FailureInfo.SET_COLLATERAL_FACTOR_VALIDATION
        }

        // If collateral factor != 0, fail if price == 0
        if (newCollateralFactorMantissa != 0 && oracle.getUnderlyingPrice(cToken) == 0) {
            return fail(Error.PRICE_ERROR, FailureInfo.SET_COLLATERAL_FACTOR_WITHOUT_PRICE);
        }

        // Set market's collateral factor to new collateral factor, remember old value
        uint oldCollateralFactorMantissa = market.collateralFactorMantissa;
        market.collateralFactorMantissa = newCollateralFactorMantissa;

        // Emit event with asset, old collateral factor, and new collateral factor
        emit NewCollateralFactor(cToken, oldCollateralFactorMantissa, newCollateralFactorMantissa);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Sets maxAssets which controls how many markets can be entered
      * @dev Admin function to set maxAssets
      * @param newMaxAssets New max assets
      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
```

```
     */
    function _setMaxAssets(uint newMaxAssets) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_MAX_ASSETS_OWNER_CHECK);
        }

        uint oldMaxAssets = maxAssets;
        maxAssets = newMaxAssets;
        emit NewMaxAssets(oldMaxAssets, newMaxAssets);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Sets liquidationIncentive
     * @dev Admin function to set liquidationIncentive
     * @param newLiquidationIncentiveMantissa New liquidationIncentive scaled by 1e18
     * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
     */
    function _setLiquidationIncentive(uint newLiquidationIncentiveMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_LIQUIDATION_INCENTIVE_OWNER_CHECK);
        }

        // Check de-scaled min <= newLiquidationIncentive <= max
        Exp memory newLiquidationIncentive = Exp({mantissa: newLiquidationIncentiveMantissa});
        Exp memory minLiquidationIncentive = Exp({mantissa: liquidationIncentiveMinMantissa});
        if (lessThanExp(newLiquidationIncentive, minLiquidationIncentive)) {
            return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VA
        }

        Exp memory maxLiquidationIncentive = Exp({mantissa: liquidationIncentiveMaxMantissa});
        if (lessThanExp(maxLiquidationIncentive, newLiquidationIncentive)) {
            return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VA
        }

        // Save current value for use in log
        uint oldLiquidationIncentiveMantissa = liquidationIncentiveMantissa;

        // Set liquidation incentive to new incentive
        liquidationIncentiveMantissa = newLiquidationIncentiveMantissa;

        // Emit event with old incentive, new incentive
        emit NewLiquidationIncentive(oldLiquidationIncentiveMantissa, newLiquidationIncentiveMantissa

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Add the market to the markets mapping and set it as listed
     * @dev Admin function to set isListed and add support for the market
     * @param cToken The address of the market (token) to list
     * @return uint 0=success, otherwise a failure. (See enum Error for details)
     */
    function _supportMarket(CToken cToken) external returns (uint) {
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
        }

        if (markets[address(cToken)].isListed) {
            return fail(Error.MARKET_ALREADY_LISTED, FailureInfo.SUPPORT_MARKET_EXISTS);
        }

        cToken.isCToken(); // Sanity check to make sure its really a CToken
        markets[address(cToken)] = Market({isListed: true, isComped: false, collateralFactorMantissa:

        _addMarketInternal(address(cToken));

        emit MarketListed(cToken);

        return uint(Error.NO_ERROR);
    }

    function _addMarketInternal(address cToken) internal {
        for (uint i = 0; i < allMarkets.length; i ++) {
            require(allMarkets[i] != CToken(cToken), "market already added");
        }
        allMarkets.push(CToken(cToken));
    }

    /**
     * @notice Admin function to change the Pause Guardian
     * @param newPauseGuardian The address of the new Pause Guardian
     * @return uint 0=success, otherwise a failure. (See enum Error for details)
     */
    function _setPauseGuardian(address newPauseGuardian) public returns (uint) {
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PAUSE_GUARDIAN_OWNER_CHECK);
        }

        // Save current value for inclusion in log
        address oldPauseGuardian = pauseGuardian;

        // Store pauseGuardian with value newPauseGuardian
        pauseGuardian = newPauseGuardian;

        // Emit NewPauseGuardian(OldPauseGuardian, NewPauseGuardian)
        emit NewPauseGuardian(oldPauseGuardian, pauseGuardian);

        return uint(Error.NO_ERROR);
    }

    function _setMintPaused(CToken cToken, bool state) public returns (bool) {
        require(markets[address(cToken)].isListed, "cannot pause a market that is not listed");
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin ca
        require(msg.sender == admin || state == true, "only admin can unpause");

        mintGuardianPaused[address(cToken)] = state;
        emit ActionPaused(cToken, "Mint", state);
        return state;
    }

    function _setBorrowPaused(CToken cToken, bool state) public returns (bool) {
        require(markets[address(cToken)].isListed, "cannot pause a market that is not listed");
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin ca
        require(msg.sender == admin || state == true, "only admin can unpause");

        borrowGuardianPaused[address(cToken)] = state;
        emit ActionPaused(cToken, "Borrow", state);
        return state;
    }

    function _setTransferPaused(bool state) public returns (bool) {
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin ca
        require(msg.sender == admin || state == true, "only admin can unpause");

        transferGuardianPaused = state;
        emit ActionPaused("Transfer", state);
        return state;
    }

    function _setSeizePaused(bool state) public returns (bool) {
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin ca
        require(msg.sender == admin || state == true, "only admin can unpause");

        seizeGuardianPaused = state;
        emit ActionPaused("Seize", state);
        return state;
    }

    function _become(Unitroller unitroller) public {
        require(msg.sender == unitroller.admin(), "only unitroller admin can change brains");
        require(unitroller._acceptImplementation() == 0, "change not authorized");
    }

    /**
     * @notice Checks caller is admin, or this contract is becoming the new implementation
     */
    function adminOrInitializing() internal view returns (bool) {
        return msg.sender == admin || msg.sender == comptrollerImplementation;
    }

    /*** Comp Distribution ***/

    /**
     * @notice Recalculate and update COMP speeds for all COMP markets
     */
    function refreshCompSpeeds() public {
        require(msg.sender == tx.origin, "only externally owned accounts may refresh speeds");
        refreshCompSpeedsInternal();
    }

    function refreshCompSpeedsInternal() internal {
        CToken[] memory allMarkets_ = allMarkets;

        for (uint i = 0; i < allMarkets_.length; i++) {
            CToken cToken = allMarkets_[i];
            Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
            updateCompSupplyIndex(address(cToken));
            updateCompBorrowIndex(address(cToken), borrowIndex);
        }

        Exp memory totalUtility = Exp({mantissa: 0});
        Exp[] memory utilities = new Exp[](allMarkets_.length);
        for (uint i = 0; i < allMarkets_.length; i++) {
            CToken cToken = allMarkets_[i];
            if (markets[address(cToken)].isComped) {
                oracle.updatePrice(cToken);
                Exp memory assetPrice = Exp({mantissa: oracle.getUnderlyingPrice(cToken)});
                Exp memory utility = mul_(assetPrice, cToken.totalBorrows());
                utilities[i] = utility;
                totalUtility = add_(totalUtility, utility);
            }
        }

        for (uint i = 0; i < allMarkets_.length; i++) {
            CToken cToken = allMarkets[i];
            uint newSpeed = totalUtility.mantissa > 0 ? mul_(compRate, div_(utilities[i], totalUtilit
            compSpeeds[address(cToken)] = newSpeed;
            emit CompSpeedUpdated(cToken, newSpeed);
        }
    }

    /**
     * @notice Accrue COMP to the market by updating the supply index
     * @param cToken The market whose supply index to update
     */
    function updateCompSupplyIndex(address cToken) internal {
        CompMarketState storage supplyState = compSupplyState[cToken];
        uint supplySpeed = compSpeeds[cToken];
        uint blockNumber = getBlockNumber();
        uint deltaBlocks = sub_(blockNumber, uint(supplyState.block));
        if (deltaBlocks > 0 && supplySpeed > 0) {
            uint supplyTokens = CToken(cToken).totalSupply();
            uint compAccrued = mul_(deltaBlocks, supplySpeed);
            Double memory ratio = supplyTokens > 0 ? fraction(compAccrued, supplyTokens) : Double({ma
            Double memory index = add_(Double({mantissa: supplyState.index}), ratio);
            compSupplyState[cToken] = CompMarketState({
            index: safe224(index.mantissa, "new index exceeds 224 bits"),
            block: safe32(blockNumber, "block number exceeds 32 bits")
            });
        } else if (deltaBlocks > 0) {
            supplyState.block = safe32(blockNumber, "block number exceeds 32 bits");
        }
    }

    /**
     * @notice Accrue COMP to the market by updating the borrow index
     * @param cToken The market whose borrow index to update
     */
    function updateCompBorrowIndex(address cToken, Exp memory marketBorrowIndex) internal {
        // CompMarketState storage borrowState = compBorrowState[cToken];
        // uint borrowSpeed = compSpeeds[cToken];
        // uint blockNumber = getBlockNumber();
        // uint deltaBlocks = sub_(blockNumber, uint(borrowState.block));
        // if (deltaBlocks > 0 && borrowSpeed > 0) {
        //     uint borrowAmount = div_(CToken(cToken).totalBorrows(), marketBorrowIndex);
        //     uint compAccrued = mul_(deltaBlocks, borrowSpeed);
        //     Double memory ratio = borrowAmount > 0 ? fraction(compAccrued, borrowAmount) : Double(
        //     Double memory index = add_(Double({mantissa: borrowState.index}), ratio);
        //     compBorrowState[cToken] = CompMarketState({
        //         index: safe224(index.mantissa, "new index exceeds 224 bits"),
        //         block: safe32(blockNumber, "block number exceeds 32 bits")
        //     });
        // } else if (deltaBlocks > 0) {
        //     borrowState.block = safe32(blockNumber, "block number exceeds 32 bits");
        // }
    }

    /**
     * @notice Calculate COMP accrued by a supplier and possibly transfer it to them
     * @param cToken The market in which the supplier is interacting
     * @param supplier The address of the supplier to distribute COMP to
     */
    function distributeSupplierComp(address cToken, address supplier, bool distributeAll) internal {
        CompMarketState storage supplyState = compSupplyState[cToken];
        Double memory supplyIndex = Double({mantissa: supplyState.index});
        Double memory supplierIndex = Double({mantissa: compSupplierIndex[cToken][supplier]});
        compSupplierIndex[cToken][supplier] = supplyIndex.mantissa;

        if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
            supplierIndex.mantissa = compInitialIndex;
        }

        Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
        uint supplierTokens = CToken(cToken).balanceOf(supplier);
        uint supplierDelta = mul_(supplierTokens, deltaIndex);
        uint supplierAccrued = add_(compAccrued[supplier], supplierDelta);
        compAccrued[supplier] = transferComp(supplier, supplierAccrued, distributeAll ? 0 : compClaim
        emit DistributedSupplierComp(CToken(cToken), supplier, supplierDelta, supplyIndex.mantissa);
    }
    /**
     * @notice Calculate COMP accrued by a borrower and possibly transfer it to them
     * @dev Borrowers will not begin to accrue until after the first interaction with the protocol.
     * @param cToken The market in which the borrower is interacting
     * @param borrower The address of the borrower to distribute COMP to
     */
    function distributeBorrowerComp(address cToken, address borrower, Exp memory marketBorrowIndex, b
        // CompMarketState storage borrowState = compBorrowState[cToken];
        // Double memory borrowIndex = Double({mantissa: borrowState.index});
        // Double memory borrowerIndex = Double({mantissa: compBorrowerIndex[cToken][borrower]});
        // compBorrowerIndex[cToken][borrower] = borrowIndex.mantissa;

        // if (borrowerIndex.mantissa > 0) {
        //     Double memory deltaIndex = sub_(borrowIndex, borrowerIndex);
        //     uint borrowerAmount = div_(CToken(cToken).borrowBalanceStored(borrower), marketBorrowI
        //     uint borrowerDelta = mul_(borrowerAmount, deltaIndex);
        //     uint borrowerAccrued = add_(compAccrued[borrower], borrowerDelta);
        //     compAccrued[borrower] = transferComp(borrower, borrowerAccrued, distributeAll ? 0 : co
        //     emit DistributedBorrowerComp(CToken(cToken), borrower, borrowerDelta, borrowIndex.mant
        // }
    }

    /**
     * @notice Transfer COMP to the user, if they are above the threshold
     * @dev Note: If there is not enough COMP, we do not perform the transfer all.
     * @param user The address of the user to transfer COMP to
     * @param userAccrued The amount of COMP to (possibly) transfer
     * @return The amount of COMP which was NOT transferred to the user
     */
    function transferComp(address user, uint userAccrued, uint threshold) internal returns (uint) {
        if (userAccrued >= threshold && userAccrued > 0) {
            Comp comp = Comp(getCompAddress());
            uint compRemaining = comp.balanceOf(address(this));
            if (userAccrued <= compRemaining) {
                comp.transfer(user, userAccrued);
                return 0;
            }
        }
        return userAccrued;
    }

    /**
     * @notice Claim all the comp accrued by holder in all markets
     * @param holder The address to claim COMP for
     */
    function claimComp(address holder) public {
        return claimComp(holder, allMarkets);
    }

    /**
     * @notice Claim all the comp accrued by holder in the specified markets
     * @param holder The address to claim COMP for
     * @param cTokens The list of markets to claim COMP in
     */
    function claimComp(address holder, CToken[] memory cTokens) public {
        address[] memory holders = new address[](1);
        holders[0] = holder;
        claimComp(holders, cTokens, true, true);
    }

    /**
     * @notice Claim all comp accrued by the holders
     * @param holders The addresses to claim COMP for
     * @param cTokens The list of markets to claim COMP in
     * @param borrowers Whether or not to claim COMP earned by borrowing
     * @param suppliers Whether or not to claim COMP earned by supplying
     */
    function claimComp(address[] memory holders, CToken[] memory cTokens, bool borrowers, bool suppli
        for (uint i = 0; i < cTokens.length; i++) {
            CToken cToken = cTokens[i];
            require(markets[address(cToken)].isListed, "market must be listed");
            if (borrowers == true) {
                Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
                updateCompBorrowIndex(address(cToken), borrowIndex);
                for (uint j = 0; j < holders.length; j++) {
                    distributeBorrowerComp(address(cToken), holders[j], borrowIndex, true);
                }
            }
            if (suppliers == true) {
                updateCompSupplyIndex(address(cToken));
                for (uint j = 0; j < holders.length; j++) {
                    distributeSupplierComp(address(cToken), holders[j], true);
                }
            }
        }
    }

    /*** Comp Distribution Admin ***/

    /**
     * @notice Set the amount of COMP distributed per block
     * @param compRate_ The amount of COMP wei per block to distribute
     */
    function _setCompRate(uint compRate_) public {
        require(adminOrInitializing(), "only admin can change comp rate");

        uint oldRate = compRate;
        compRate = compRate_;
        emit NewCompRate(oldRate, compRate_);

        refreshCompSpeedsInternal();
    }

    /**
     * @notice Add markets to compMarkets, allowing them to earn COMP in the flywheel
     * @param cTokens The addresses of the markets to add
     */
    function _addCompMarkets(address[] memory cTokens) public {
        require(adminOrInitializing(), "only admin can add comp market");

        for (uint i = 0; i < cTokens.length; i++) {
            _addCompMarketInternal(cTokens[i]);
        }

        refreshCompSpeedsInternal();
    }

    function _addCompMarketInternal(address cToken) internal {
        Market storage market = markets[cToken];
        require(market.isListed == true, "comp market is not listed");
        require(market.isComped == false, "comp market already added");

        market.isComped = true;
        emit MarketComped(CToken(cToken), true);

        if (compSupplyState[cToken].index == 0 && compSupplyState[cToken].block == 0) {
            compSupplyState[cToken] = CompMarketState({
            index: compInitialIndex,
            block: safe32(getBlockNumber(), "block number exceeds 32 bits")
            });
        }
        // if (compBorrowState[cToken].index == 0 && compBorrowState[cToken].block == 0) {
        //     compBorrowState[cToken] = CompMarketState({
        //         index: compInitialIndex,
        //         block: safe32(getBlockNumber(), "block number exceeds 32 bits")
        //     });
        // }
    }

    /**
     * @notice Remove a market from compMarkets, preventing it from earning COMP in the flywheel
     * @param cToken The address of the market to drop
     */
    function _dropCompMarket(address cToken) public {
        require(msg.sender == admin, "only admin can drop comp market");

        Market storage market = markets[cToken];
        require(market.isComped == true, "market is not a comp market");

        market.isComped = false;
        emit MarketComped(CToken(cToken), false);

        refreshCompSpeedsInternal();
    }

    /**
     * @notice Return all of the markets
     * @dev The automatic getter may be used to access an individual market.
     * @return The list of market addresses
     */
    function getAllMarkets() public view returns (CToken[] memory) {
        return allMarkets;
    }

    function getBlockNumber() public view returns (uint) {
        return block.number;
    }

    function setCompAddress(address _compAddress) external {
        require(msg.sender == admin, "You are not an admin");
        compAddress = _compAddress;
    }

    /**
     * @notice Return the address of the COMP token
     * @return The address of COMP
     */
    function getCompAddress() public view returns (address) {
        return (compAddress);
    }
}



pragma solidity ^0.5.16;
import "./CToken.sol";
import "./ErrorReporter.sol";

contract ComptrollerInterface {
    /// @notice Indicator that this is a Comptroller contract (for inspection)
    bool public constant isComptroller = true;

    /*** Assets You Are In ***/

    function enterMarkets(address[] calldata cTokens)
    external
    returns (uint256[] memory);

    function exitMarket(address cToken) external returns (uint256);

    function addToMarketExternal(address cToken, address underlying, address borrower)
    external;

    /*** Policy Hooks ***/

    function mintAllowed(
        address cToken,
        address minter,
        uint256 mintAmount
    ) external returns (uint256);

    function mintVerify(
        address cToken,
        address minter,
        uint256 mintAmount,
        uint256 mintTokens
    ) external;

    function redeemAllowed(
        address cToken,
        address redeemer,
        uint256 redeemTokens
    ) external returns (uint256);

    function redeemVerify(
        address cToken,
        address redeemer,
        uint256 redeemAmount,
        uint256 redeemTokens
    ) external;

    function borrowAllowed(
        address cToken,
        address borrower,
        uint256 borrowAmount
    ) external returns (uint256);

    function borrowVerify(
        address cToken,
        address borrower,
        uint256 borrowAmount
    ) external;

    function repayBorrowAllowed(
        address cToken,
        address payer,
        address borrower,
        uint256 repayAmount
    ) external returns (uint256);

    function repayBorrowVerify(
        address cToken,
        address payer,
        address borrower,
        uint256 repayAmount,
        uint256 borrowerIndex
    ) external;

    function liquidateBorrowAllowed(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint256 repayAmount
    ) external returns (uint256);

    function liquidateBorrowVerify(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint256 repayAmount,
        uint256 seizeTokens
    ) external;

    function seizeAllowed(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint256 seizeTokens
    ) external returns (uint256);

    function seizeVerify(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint256 seizeTokens
    ) external;

    function transferAllowed(
        address cToken,
        address src,
        address dst,
        uint256 transferTokens
    ) external returns (uint256);

    function transferVerify(
        address cToken,
        address src,
        address dst,
        uint256 transferTokens
    ) external;

    /*** Liquidity/Liquidation Calculations ***/

    function liquidateCalculateSeizeTokens(
        address cTokenBorrowed,
        address cTokenCollateral,
        uint256 repayAmount
    ) external view returns (uint256, uint256);
}


pragma solidity ^0.5.16;

import "./CToken.sol";
import "./PriceOracle.sol";

contract UnitrollerAdminStorage {
    /**
    * @notice Administrator for this contract
    */
    address public admin;

    /**
    * @notice Pending administrator for this contract
    */
    address public pendingAdmin;

    /**
    * @notice Active brains of Unitroller
    */
    address public comptrollerImplementation;

    /**
    * @notice Pending brains of Unitroller
    */
    address public pendingComptrollerImplementation;
}

contract ComptrollerV1Storage is UnitrollerAdminStorage {

    /**
     * @notice Oracle which gives the price of any given asset
     */
    PriceOracle public oracle;

    /**
     * @notice Multiplier used to calculate the maximum repayAmount when liquidating a borrow
     */
    uint public closeFactorMantissa;

    /**
     * @notice Multiplier representing the discount on collateral that a liquidator receives
     */
    uint public liquidationIncentiveMantissa;

    /**
     * @notice Max number of assets a single account can participate in (borrow or use as collateral)
     */
    uint public maxAssets;

    /**
     * @notice Per-account mapping of "assets you are in", capped by maxAssets
     */
    mapping(address => CToken[]) public accountAssets;

}

contract ComptrollerV2Storage is ComptrollerV1Storage {
    struct Market {
        /// @notice Whether or not this market is listed
        bool isListed;

        /**
         * @notice Multiplier representing the most one can borrow against their collateral in this m
         *  For instance, 0.9 to allow borrowing 90% of collateral value.
         *  Must be between 0 and 1, and stored as a mantissa.
         */
        uint collateralFactorMantissa;

        /// @notice Per-market mapping of "accounts in this asset"
        mapping(address => bool) accountMembership;

        /// @notice Whether or not this market receives COMP
        bool isComped;
    }

    /**
     * @notice Official mapping of cTokens -> Market metadata
     * @dev Used e.g. to determine if a market is supported
     */
    mapping(address => Market) public markets;


    /**
     * @notice The Pause Guardian can pause certain actions as a safety mechanism.
     *  Actions which allow users to remove their own assets cannot be paused.
     *  Liquidation / seizing / transfer can only be paused globally, not by market.
     */
    address public pauseGuardian;
    bool public _mintGuardianPaused;
    bool public _borrowGuardianPaused;
    bool public transferGuardianPaused;
    bool public seizeGuardianPaused;
    mapping(address => bool) public mintGuardianPaused;
    mapping(address => bool) public borrowGuardianPaused;
}

contract ComptrollerV3Storage is ComptrollerV2Storage {
    struct CompMarketState {
        /// @notice The market's last updated compBorrowIndex or compSupplyIndex
        uint224 index;

        /// @notice The block number the index was last updated at
        uint32 block;
    }

    /// @notice A list of all markets
    CToken[] public allMarkets;

    /// @notice The rate at which the flywheel distributes COMP, per block
    uint public compRate;

    /// @notice The portion of compRate that each market currently receives
    mapping(address => uint) public compSpeeds;

    /// @notice The COMP market supply state for each market
    mapping(address => CompMarketState) public compSupplyState;

    /// @notice The COMP market borrow state for each market
    mapping(address => CompMarketState) public compBorrowState;

    /// @notice The COMP borrow index for each market for each supplier as of the last time they accr
    mapping(address => mapping(address => uint)) public compSupplierIndex;

    /// @notice The COMP borrow index for each market for each borrower as of the last time they accr
    mapping(address => mapping(address => uint)) public compBorrowerIndex;

    /// @notice The COMP accrued but not yet transferred to each user
    mapping(address => uint) public compAccrued;
}



pragma solidity ^0.5.16;

/**
 * @title ERC 20 Token Standard Interface
 *  https://eips.ethereum.org/EIPS/eip-20
 */
interface EIP20Interface {
    function name() external view returns (string memory);
    function symbol() external view returns (string memory);
    function decimals() external view returns (uint8);

    /**
```

```
       * @notice Get the total number of tokens in circulation
       * @return The supply of tokens
       */
    function totalSupply() external view returns (uint256);

    /**
     * @notice Gets the balance of the specified address
     * @param owner The address from which the balance will be retrieved
     * @return The balance
     */
    function balanceOf(address owner) external view returns (uint256 balance);

    /**
      * @notice Transfer `amount` tokens from `msg.sender` to `dst`
      * @param dst The address of the destination account
      * @param amount The number of tokens to transfer
      * @return Whether or not the transfer succeeded
      */
    function transfer(address dst, uint256 amount) external returns (bool success);

    /**
      * @notice Transfer `amount` tokens from `src` to `dst`
      * @param src The address of the source account
      * @param dst The address of the destination account
      * @param amount The number of tokens to transfer
      * @return Whether or not the transfer succeeded
      */
    function transferFrom(address src, address dst, uint256 amount) external returns (bool success);

    /**
      * @notice Approve `spender` to transfer up to `amount` from `src`
      * @dev This will overwrite the approval amount for `spender`
      *  and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
      * @param spender The address of the account which may transfer tokens
      * @param amount The number of tokens that are approved (-1 means infinite)
      * @return Whether or not the approval succeeded
      */
    function approve(address spender, uint256 amount) external returns (bool success);

    /**
      * @notice Get the current allowance from `owner` for `spender`
      * @param owner The address of the account which owns the tokens to be spent
      * @param spender The address of the account which may transfer tokens
      * @return The number of tokens allowed to be spent (-1 means infinite)
      */
    function allowance(address owner, address spender) external view returns (uint256 remaining);

    event Transfer(address indexed from, address indexed to, uint256 amount);
    event Approval(address indexed owner, address indexed spender, uint256 amount);
}



pragma solidity ^0.5.16;

/**
 * @title EIP20NonStandardInterface
 * @dev Version of ERC20 with no return values for `transfer` and `transferFrom`
 *  See https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521
 */
interface EIP20NonStandardInterface {

    /**
     * @notice Get the total number of tokens in circulation
     * @return The supply of tokens
     */
    function totalSupply() external view returns (uint256);

    /**
     * @notice Gets the balance of the specified address
     * @param owner The address from which the balance will be retrieved
     * @return The balance
     */
    function balanceOf(address owner) external view returns (uint256 balance);

    ///
    /// !!!!!!!!!!!!!!
    /// !!! NOTICE !!! `transfer` does not return a value, in violation of the ERC-20 specification
    /// !!!!!!!!!!!!!!
    ///

    /**
      * @notice Transfer `amount` tokens from `msg.sender` to `dst`
      * @param dst The address of the destination account
      * @param amount The number of tokens to transfer
      */
    function transfer(address dst, uint256 amount) external;

    ///
    /// !!!!!!!!!!!!!!
    /// !!! NOTICE !!! `transferFrom` does not return a value, in violation of the ERC-20 specificati
    /// !!!!!!!!!!!!!!
    ///

    /**
      * @notice Transfer `amount` tokens from `src` to `dst`
      * @param src The address of the source account
      * @param dst The address of the destination account
      * @param amount The number of tokens to transfer
      */
    function transferFrom(address src, address dst, uint256 amount) external;

    /**
      * @notice Approve `spender` to transfer up to `amount` from `src`
      * @dev This will overwrite the approval amount for `spender`
      *  and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
      * @param spender The address of the account which may transfer tokens
      * @param amount The number of tokens that are approved
      * @return Whether or not the approval succeeded
      */
    function approve(address spender, uint256 amount) external returns (bool success);

    /**
      * @notice Get the current allowance from `owner` for `spender`
      * @param owner The address of the account which owns the tokens to be spent
      * @param spender The address of the account which may transfer tokens
      * @return The number of tokens allowed to be spent
      */
    function allowance(address owner, address spender) external view returns (uint256 remaining);

    event Transfer(address indexed from, address indexed to, uint256 amount);
    event Approval(address indexed owner, address indexed spender, uint256 amount);
}



pragma solidity ^0.5.16;

contract ComptrollerErrorReporter {
    enum Error {
        NO_ERROR,
        UNAUTHORIZED,
        COMPTROLLER_MISMATCH,
        INSUFFICIENT_SHORTFALL,
        INSUFFICIENT_LIQUIDITY,
        INVALID_CLOSE_FACTOR,
        INVALID_COLLATERAL_FACTOR,
        INVALID_LIQUIDATION_INCENTIVE,
        MARKET_NOT_ENTERED, // no longer possible
        MARKET_NOT_LISTED,
        MARKET_ALREADY_LISTED,
        MATH_ERROR,
        NONZERO_BORROW_BALANCE,
        PRICE_ERROR,
        REJECTION,
        SNAPSHOT_ERROR,
        TOO_MANY_ASSETS,
        TOO_MUCH_REPAY
    }

    enum FailureInfo {
        ACCEPT_ADMIN_PENDING_ADMIN_CHECK,
        ACCEPT_PENDING_IMPLEMENTATION_ADDRESS_CHECK,
        EXIT_MARKET_BALANCE_OWED,
        EXIT_MARKET_REJECTION,
        SET_CLOSE_FACTOR_OWNER_CHECK,
        SET_CLOSE_FACTOR_VALIDATION,
        SET_COLLATERAL_FACTOR_OWNER_CHECK,
        SET_COLLATERAL_FACTOR_NO_EXISTS,
        SET_COLLATERAL_FACTOR_VALIDATION,
        SET_COLLATERAL_FACTOR_WITHOUT_PRICE,
        SET_IMPLEMENTATION_OWNER_CHECK,
        SET_LIQUIDATION_INCENTIVE_OWNER_CHECK,
        SET_LIQUIDATION_INCENTIVE_VALIDATION,
        SET_MAX_ASSETS_OWNER_CHECK,
        SET_PENDING_ADMIN_OWNER_CHECK,
        SET_PENDING_IMPLEMENTATION_OWNER_CHECK,
        SET_PRICE_ORACLE_OWNER_CHECK,
        SUPPORT_MARKET_EXISTS,
        SUPPORT_MARKET_OWNER_CHECK,
        SET_PAUSE_GUARDIAN_OWNER_CHECK
    }

    /**
      * @dev `error` corresponds to enum Error; `info` corresponds to enum FailureInfo, and `detail`
      * contract-specific code that enables us to report opaque error codes from upgradeable contract
      **/
    event Failure(uint error, uint info, uint detail);

    /**
      * @dev use this when reporting a known error from the money market or a non-upgradeable collabo
      */
    function fail(Error err, FailureInfo info) internal returns (uint) {
        emit Failure(uint(err), uint(info), 0);

        return uint(err);
    }

    /**
      * @dev use this when reporting an opaque error from an upgradeable collaborator contract
      */
    function failOpaque(Error err, FailureInfo info, uint opaqueError) internal returns (uint) {
        emit Failure(uint(err), uint(info), opaqueError);

        return uint(err);
    }
}
```

```
contract TokenErrorReporter {
    enum Error {
        NO_ERROR,
        UNAUTHORIZED,
        BAD_INPUT,
        COMPTROLLER_REJECTION,
        COMPTROLLER_CALCULATION_ERROR,
        INTEREST_RATE_MODEL_ERROR,
        INVALID_ACCOUNT_PAIR,
        INVALID_CLOSE_AMOUNT_REQUESTED,
        INVALID_COLLATERAL_FACTOR,
        MATH_ERROR,
        MARKET_NOT_FRESH,
        MARKET_NOT_LISTED,
        TOKEN_INSUFFICIENT_ALLOWANCE,
        TOKEN_INSUFFICIENT_BALANCE,
        TOKEN_INSUFFICIENT_CASH,
        TOKEN_TRANSFER_IN_FAILED,
        TOKEN_TRANSFER_OUT_FAILED
    }

    /*
     * Note: FailureInfo (but not Error) is kept in alphabetical order
     *       This is because FailureInfo grows significantly faster, and
     *       the order of Error has some meaning, while the order of FailureInfo
     *       is entirely arbitrary.
     */
    enum FailureInfo {
        ACCEPT_ADMIN_PENDING_ADMIN_CHECK,
        ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED,
        ACCRUE_INTEREST_BORROW_RATE_CALCULATION_FAILED,
        ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULATION_FAILED,
        ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED,
        ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED,
        ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CALCULATION_FAILED,
        BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
        BORROW_ACCRUE_INTEREST_FAILED,
        BORROW_CASH_NOT_AVAILABLE,
        BORROW_FRESHNESS_CHECK,
        BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED,
        BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED,
        BORROW_MARKET_NOT_LISTED,
        BORROW_COMPTROLLER_REJECTION,
        LIQUIDATE_ACCRUE_BORROW_INTEREST_FAILED,
        LIQUIDATE_ACCRUE_COLLATERAL_INTEREST_FAILED,
        LIQUIDATE_COLLATERAL_FRESHNESS_CHECK,
        LIQUIDATE_COMPTROLLER_REJECTION,
        LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED,
        LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX,
        LIQUIDATE_CLOSE_AMOUNT_IS_ZERO,
        LIQUIDATE_FRESHNESS_CHECK,
        LIQUIDATE_LIQUIDATOR_IS_BORROWER,
        LIQUIDATE_REPAY_BORROW_FRESH_FAILED,
        LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED,
        LIQUIDATE_SEIZE_BALANCE_DECREMENT_FAILED,
        LIQUIDATE_SEIZE_COMPTROLLER_REJECTION,
        LIQUIDATE_SEIZE_LIQUIDATOR_IS_BORROWER,
        LIQUIDATE_SEIZE_TOO_MUCH,
        MINT_ACCRUE_INTEREST_FAILED,
        MINT_COMPTROLLER_REJECTION,
        MINT_EXCHANGE_CALCULATION_FAILED,
        MINT_EXCHANGE_RATE_READ_FAILED,
        MINT_FRESHNESS_CHECK,
        MINT_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED,
        MINT_NEW_TOTAL_SUPPLY_CALCULATION_FAILED,
        MINT_TRANSFER_IN_FAILED,
        MINT_TRANSFER_IN_NOT_POSSIBLE,
        REDEEM_ACCRUE_INTEREST_FAILED,
        REDEEM_COMPTROLLER_REJECTION,
        REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED,
        REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED,
        REDEEM_EXCHANGE_RATE_READ_FAILED,
        REDEEM_FRESHNESS_CHECK,
        REDEEM_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED,
        REDEEM_NEW_TOTAL_SUPPLY_CALCULATION_FAILED,
        REDEEM_TRANSFER_OUT_NOT_POSSIBLE,
        REDUCE_RESERVES_ACCRUE_INTEREST_FAILED,
        REDUCE_RESERVES_ADMIN_CHECK,
        REDUCE_RESERVES_CASH_NOT_AVAILABLE,
        REDUCE_RESERVES_FRESH_CHECK,
        REDUCE_RESERVES_VALIDATION,
        REPAY_BEHALF_ACCRUE_INTEREST_FAILED,
        REPAY_BORROW_ACCRUE_INTEREST_FAILED,
        REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
        REPAY_BORROW_COMPTROLLER_REJECTION,
        REPAY_BORROW_FRESHNESS_CHECK,
        REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED,
        REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED,
        REPAY_BORROW_TRANSFER_IN_NOT_POSSIBLE,
        SET_COLLATERAL_FACTOR_OWNER_CHECK,
        SET_COLLATERAL_FACTOR_VALIDATION,
        SET_COMPTROLLER_OWNER_CHECK,
        SET_INTEREST_RATE_MODEL_ACCRUE_INTEREST_FAILED,
        SET_INTEREST_RATE_MODEL_FRESH_CHECK,
        SET_INTEREST_RATE_MODEL_OWNER_CHECK,
        SET_MAX_ASSETS_OWNER_CHECK,
        SET_ORACLE_MARKET_NOT_LISTED,
        SET_PENDING_ADMIN_OWNER_CHECK,
        SET_RESERVE_FACTOR_ACCRUE_INTEREST_FAILED,
        SET_RESERVE_FACTOR_ADMIN_CHECK,
        SET_RESERVE_FACTOR_FRESH_CHECK,
        SET_RESERVE_FACTOR_BOUNDS_CHECK,
        TRANSFER_COMPTROLLER_REJECTION,
        TRANSFER_NOT_ALLOWED,
        TRANSFER_NOT_ENOUGH,
        TRANSFER_TOO_MUCH,
        ADD_RESERVES_ACCRUE_INTEREST_FAILED,
        ADD_RESERVES_FRESH_CHECK,
        ADD_RESERVES_TRANSFER_IN_NOT_POSSIBLE
    }

    /**
      * @dev `error` corresponds to enum Error; `info` corresponds to enum FailureInfo, and `detail`
      * contract-specific code that enables us to report opaque error codes from upgradeable contract
      **/
    event Failure(uint error, uint info, uint detail);

    /**
      * @dev use this when reporting a known error from the money market or a non-upgradeable collabo
      */
    function fail(Error err, FailureInfo info) internal returns (uint) {
        emit Failure(uint(err), uint(info), 0);

        return uint(err);
    }

    /**
      * @dev use this when reporting an opaque error from an upgradeable collaborator contract
      */
    function failOpaque(Error err, FailureInfo info, uint opaqueError) internal returns (uint) {
        emit Failure(uint(err), uint(info), opaqueError);
        return uint(err);
    }
}


pragma solidity ^0.5.16;

import "./CarefulMath.sol";

/**
 * @title Exponential module for storing fixed-precision decimals
 * @author Compound
 * @notice Exp is a struct which stores decimals with a fixed precision of 18 decimal places.
 *         Thus, if we wanted to store the 5.1, mantissa would store 5.1e18. That is:
 *         `Exp({mantissa: 5100000000000000000})`.
 */
contract Exponential is CarefulMath {
    uint constant expScale = 1e18;
    uint constant doubleScale = 1e36;
    uint constant halfExpScale = expScale/2;
    uint constant mantissaOne = expScale;

    struct Exp {
        uint mantissa;
    }

    struct Double {
        uint mantissa;
    }

    /**
     * @dev Creates an exponential from numerator and denominator values.
     *      Note: Returns an error if (`num` * 10e18) > MAX_INT,
     *            or if `denom` is zero.
     */
    function getExp(uint num, uint denom) pure internal returns (MathError, Exp memory) {
        (MathError err0, uint scaledNumerator) = mulUInt(num, expScale);
        if (err0 != MathError.NO_ERROR) {
            return (err0, Exp({mantissa: 0}));
        }

        (MathError err1, uint rational) = divUInt(scaledNumerator, denom);
        if (err1 != MathError.NO_ERROR) {
            return (err1, Exp({mantissa: 0}));
        }

        return (MathError.NO_ERROR, Exp({mantissa: rational}));
    }

    /**
     * @dev Adds two exponentials, returning a new exponential.
     */
    function addExp(Exp memory a, Exp memory b) pure internal returns (MathError, Exp memory) {
        (MathError error, uint result) = addUInt(a.mantissa, b.mantissa);

        return (error, Exp({mantissa: result}));
    }

    /**
     * @dev Subtracts two exponentials, returning a new exponential.
     */
    function subExp(Exp memory a, Exp memory b) pure internal returns (MathError, Exp memory) {
        (MathError error, uint result) = subUInt(a.mantissa, b.mantissa);

        return (error, Exp({mantissa: result}));
    }
```

```
/**
 * @dev Multiply an Exp by a scalar, returning a new Exp.
 */
function mulScalar(Exp memory a, uint scalar) pure internal returns (MathError, Exp memory) {
    (MathError err0, uint scaledMantissa) = mulUInt(a.mantissa, scalar);
    if (err0 != MathError.NO_ERROR) {
        return (err0, Exp({mantissa: 0}));
    }

    return (MathError.NO_ERROR, Exp({mantissa: scaledMantissa}));
}

/**
 * @dev Multiply an Exp by a scalar, then truncate to return an unsigned integer.
 */
function mulScalarTruncate(Exp memory a, uint scalar) pure internal returns (MathError, uint) {
    (MathError err, Exp memory product) = mulScalar(a, scalar);
    if (err != MathError.NO_ERROR) {
        return (err, 0);
    }

    return (MathError.NO_ERROR, truncate(product));
}

/**
 * @dev Multiply an Exp by a scalar, truncate, then add an to an unsigned integer, returning an u
 */
function mulScalarTruncateAddUInt(Exp memory a, uint scalar, uint addend) pure internal returns (
    (MathError err, Exp memory product) = mulScalar(a, scalar);
    if (err != MathError.NO_ERROR) {
        return (err, 0);
    }

    return addUInt(truncate(product), addend);
}

/**
 * @dev Divide an Exp by a scalar, returning a new Exp.
 */
function divScalar(Exp memory a, uint scalar) pure internal returns (MathError, Exp memory) {
    (MathError err0, uint descaledMantissa) = divUInt(a.mantissa, scalar);
    if (err0 != MathError.NO_ERROR) {
        return (err0, Exp({mantissa: 0}));
    }

    return (MathError.NO_ERROR, Exp({mantissa: descaledMantissa}));
}

/**
 * @dev Divide a scalar by an Exp, returning a new Exp.
 */
function divScalarByExp(uint scalar, Exp memory divisor) pure internal returns (MathError, Exp me
    /*
      We are doing this as:
      getExp(mulUInt(expScale, scalar), divisor.mantissa)

      How it works:
      Exp = a / b;
      Scalar = s;
      `s / (a / b)` = `b * s / a` and since for an Exp `a = mantissa, b = expScale`
    */
    (MathError err0, uint numerator) = mulUInt(expScale, scalar);
    if (err0 != MathError.NO_ERROR) {
        return (err0, Exp({mantissa: 0}));
    }
```

```
        return getExp(numerator, divisor.mantissa);
    }

    /**
     * @dev Divide a scalar by an Exp, then truncate to return an unsigned integer.
     */
    function divScalarByExpTruncate(uint scalar, Exp memory divisor) pure internal returns (MathError
        (MathError err, Exp memory fraction) = divScalarByExp(scalar, divisor);
        if (err != MathError.NO_ERROR) {
            return (err, 0);
        }

        return (MathError.NO_ERROR, truncate(fraction));
    }

    /**
     * @dev Multiplies two exponentials, returning a new exponential.
     */
    function mulExp(Exp memory a, Exp memory b) pure internal returns (MathError, Exp memory) {

        (MathError err0, uint doubleScaledProduct) = mulUInt(a.mantissa, b.mantissa);
        if (err0 != MathError.NO_ERROR) {
            return (err0, Exp({mantissa: 0}));
        }

        // We add half the scale before dividing so that we get rounding instead of truncation.
        //  See "Listing 6" and text above it at https://accu.org/index.php/journals/1717
        // Without this change, a result like 6.6...e-19 will be truncated to 0 instead of being roun
        (MathError err1, uint doubleScaledProductWithHalfScale) = addUInt(halfExpScale, doubleScaledP
        if (err1 != MathError.NO_ERROR) {
            return (err1, Exp({mantissa: 0}));
        }

        (MathError err2, uint product) = divUInt(doubleScaledProductWithHalfScale, expScale);
        // The only error `div` can return is MathError.DIVISION_BY_ZERO but we control `expScale` an
        assert(err2 == MathError.NO_ERROR);

        return (MathError.NO_ERROR, Exp({mantissa: product}));
    }

    /**
     * @dev Multiplies two exponentials given their mantissas, returning a new exponential.
     */
    function mulExp(uint a, uint b) pure internal returns (MathError, Exp memory) {
        return mulExp(Exp({mantissa: a}), Exp({mantissa: b}));
    }

    /**
     * @dev Multiplies three exponentials, returning a new exponential.
     */
    function mulExp3(Exp memory a, Exp memory b, Exp memory c) pure internal returns (MathError, Exp
        (MathError err, Exp memory ab) = mulExp(a, b);
        if (err != MathError.NO_ERROR) {
            return (err, ab);
        }
        return mulExp(ab, c);
    }

    /**
     * @dev Divides two exponentials, returning a new exponential.
     *     (a/scale) / (b/scale) = (a/scale) * (scale/b) = a/b,
     *  which we can scale as an Exp by calling getExp(a.mantissa, b.mantissa)
     */
    function divExp(Exp memory a, Exp memory b) pure internal returns (MathError, Exp memory) {
        return getExp(a.mantissa, b.mantissa);
    }
```

```
/**
 * @dev Truncates the given exp to a whole number value.
 *      For example, truncate(Exp{mantissa: 15 * expScale}) = 15
 */
function truncate(Exp memory exp) pure internal returns (uint) {
    // Note: We are not using careful math here as we're performing a division that cannot fail
    return exp.mantissa / expScale;
}

/**
 * @dev Checks if first Exp is less than second Exp.
 */
function lessThanExp(Exp memory left, Exp memory right) pure internal returns (bool) {
    return left.mantissa < right.mantissa;
}

/**
 * @dev Checks if left Exp <= right Exp.
 */
function lessThanOrEqualExp(Exp memory left, Exp memory right) pure internal returns (bool) {
    return left.mantissa <= right.mantissa;
}

/**
 * @dev Checks if left Exp > right Exp.
 */
function greaterThanExp(Exp memory left, Exp memory right) pure internal returns (bool) {
    return left.mantissa > right.mantissa;
}

/**
 * @dev returns true if Exp is exactly zero
 */
function isZeroExp(Exp memory value) pure internal returns (bool) {
    return value.mantissa == 0;
}

function safe224(uint n, string memory errorMessage) pure internal returns (uint224) {
    require(n < 2**224, errorMessage);
    return uint224(n);
}

function safe32(uint n, string memory errorMessage) pure internal returns (uint32) {
    require(n < 2**32, errorMessage);
    return uint32(n);
}

function add_(Exp memory a, Exp memory b) pure internal returns (Exp memory) {
    return Exp({mantissa: add_(a.mantissa, b.mantissa)});
}

function add_(Double memory a, Double memory b) pure internal returns (Double memory) {
    return Double({mantissa: add_(a.mantissa, b.mantissa)});
}

function add_(uint a, uint b) pure internal returns (uint) {
    return add_(a, b, "addition overflow");
}

function add_(uint a, uint b, string memory errorMessage) pure internal returns (uint) {
    uint c = a + b;
    require(c >= a, errorMessage);
    return c;
}
```

```
    function sub_(Exp memory a, Exp memory b) pure internal returns (Exp memory) {
        return Exp({mantissa: sub_(a.mantissa, b.mantissa)});
    }

    function sub_(Double memory a, Double memory b) pure internal returns (Double memory) {
        return Double({mantissa: sub_(a.mantissa, b.mantissa)});
    }

    function sub_(uint a, uint b) pure internal returns (uint) {
        return sub_(a, b, "subtraction underflow");
    }

    function sub_(uint a, uint b, string memory errorMessage) pure internal returns (uint) {
        require(b <= a, errorMessage);
        return a - b;
    }

    function mul_(Exp memory a, Exp memory b) pure internal returns (Exp memory) {
        return Exp({mantissa: mul_(a.mantissa, b.mantissa) / expScale});
    }

    function mul_(Exp memory a, uint b) pure internal returns (Exp memory) {
        return Exp({mantissa: mul_(a.mantissa, b)});
    }

    function mul_(uint a, Exp memory b) pure internal returns (uint) {
        return mul_(a, b.mantissa) / expScale;
    }

    function mul_(Double memory a, Double memory b) pure internal returns (Double memory) {
        return Double({mantissa: mul_(a.mantissa, b.mantissa) / doubleScale});
    }

    function mul_(Double memory a, uint b) pure internal returns (Double memory) {
        return Double({mantissa: mul_(a.mantissa, b)});
    }

    function mul_(uint a, Double memory b) pure internal returns (uint) {
        return mul_(a, b.mantissa) / doubleScale;
    }

    function mul_(uint a, uint b) pure internal returns (uint) {
        return mul_(a, b, "multiplication overflow");
    }

    function mul_(uint a, uint b, string memory errorMessage) pure internal returns (uint) {
        if (a == 0 || b == 0) {
            return 0;
        }
        uint c = a * b;
        require(c / a == b, errorMessage);
        return c;
    }

    function div_(Exp memory a, Exp memory b) pure internal returns (Exp memory) {
        return Exp({mantissa: div_(mul_(a.mantissa, expScale), b.mantissa)});
    }

    function div_(Exp memory a, uint b) pure internal returns (Exp memory) {
        return Exp({mantissa: div_(a.mantissa, b)});
    }

    function div_(uint a, Exp memory b) pure internal returns (uint) {
        return div_(mul_(a, expScale), b.mantissa);
    }
    function div_(Double memory a, Double memory b) pure internal returns (Double memory) {
        return Double({mantissa: div_(mul_(a.mantissa, doubleScale), b.mantissa)});
    }

    function div_(Double memory a, uint b) pure internal returns (Double memory) {
        return Double({mantissa: div_(a.mantissa, b)});
    }

    function div_(uint a, Double memory b) pure internal returns (uint) {
        return div_(mul_(a, doubleScale), b.mantissa);
    }

    function div_(uint a, uint b) pure internal returns (uint) {
        return div_(a, b, "divide by zero");
    }

    function div_(uint a, uint b, string memory errorMessage) pure internal returns (uint) {
        require(b > 0, errorMessage);
        return a / b;
    }

    function fraction(uint a, uint b) pure internal returns (Double memory) {
        return Double({mantissa: div_(mul_(a, doubleScale), b)});
    }
}



pragma solidity ^0.5.16;

/**
  * @title Compound's InterestRateModel Interface
  * @author Compound
  */
contract InterestRateModel {
    /// @notice Indicator that this is an InterestRateModel contract (for inspection)
    bool public constant isInterestRateModel = true;

    /**
      * @notice Calculates the current borrow interest rate per block
      * @param cash The total amount of cash the market has
      * @param borrows The total amount of borrows the market has outstanding
      * @param reserves The total amnount of reserves the market has
      * @return The borrow rate per block (as a percentage, and scaled by 1e18)
      */
    function getBorrowRate(uint cash, uint borrows, uint reserves) external view returns (uint);

    /**
      * @notice Calculates the current supply interest rate per block
      * @param cash The total amount of cash the market has
      * @param borrows The total amount of borrows the market has outstanding
      * @param reserves The total amnount of reserves the market has
      * @param reserveFactorMantissa The current reserve factor the market has
      * @return The supply rate per block (as a percentage, and scaled by 1e18)
      */
    function getSupplyRate(uint cash, uint borrows, uint reserves, uint reserveFactorMantissa) extern

}



import "./CToken.sol";

pragma solidity ^0.5.16;

contract MockPriceOracle {
    /// @notice Indicator that this is a PriceOracle contract (for inspection)
    bool public constant isPriceOracle = true;
    uint value;

    mapping(address => uint) public prices;

    // this is a stub for the price oracle interface
    function updatePrice(CToken cToken) external {

    }

    /**
      * @notice Update the price of an underlying asset
      * @param cToken The cToken to update the underlying price of
      */
    function mockUpdatePrice(address cToken, uint price) external{
        prices[cToken] = price;
    }

    /**
      * @notice Get the underlying price of a cToken asset
      * @param cToken The cToken to get the underlying price of
      * @return The underlying asset price mantissa (scaled by 1e18).
      *  Zero means the price is unavailable.
      */
    function getUnderlyingPrice(address cToken) external view returns (uint){
        return prices[cToken];
    }
}



pragma solidity ^0.5.16;

import "./CToken.sol";

contract PriceOracle {

    /// @notice Indicator that this is a PriceOracle contract (for inspection)
    bool public constant isPriceOracle = true;

    /**
    * @notice Update the price of an underlying asset
* @param cToken The cToken to update the underlying price of
*/

    function updatePrice(CToken cToken) external;

    /**
    * @notice Get the underlying price of a cToken asset
* @param cToken The cToken to get the underlying price of
* @return The underlying asset price mantissa (scaled by 1e18).
* Zero means the price is unavailable.
*/

    function getUnderlyingPrice(CToken cToken) external view returns (uint);

}


pragma solidity ^0.5.16;

import "./ErrorReporter.sol";
import "./ComptrollerStorage.sol";
/**
 * @title ComptrollerCore
 * @dev Storage for the comptroller is at this address, while execution is delegated to the `comptrol
 * CTokens should reference this contract as their comptroller.
 */
contract Unitroller is UnitrollerAdminStorage, ComptrollerErrorReporter {

    /**
      * @notice Emitted when pendingComptrollerImplementation is changed
      */
    event NewPendingImplementation(address oldPendingImplementation, address newPendingImplementation

    /**
      * @notice Emitted when pendingComptrollerImplementation is accepted, which means comptroller im
      */
    event NewImplementation(address oldImplementation, address newImplementation);

    /**
      * @notice Emitted when pendingAdmin is changed
      */
    event NewPendingAdmin(address oldPendingAdmin, address newPendingAdmin);

    /**
      * @notice Emitted when pendingAdmin is accepted, which means admin is updated
      */
    event NewAdmin(address oldAdmin, address newAdmin);

    constructor() public {
        // Set admin to caller
        admin = msg.sender;
    }

    /*** Admin Functions ***/
    function _setPendingImplementation(address newPendingImplementation) public returns (uint) {

        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PENDING_IMPLEMENTATION_OWNER_CHECK);
        }

        address oldPendingImplementation = pendingComptrollerImplementation;

        pendingComptrollerImplementation = newPendingImplementation;

        emit NewPendingImplementation(oldPendingImplementation, pendingComptrollerImplementation);

        return uint(Error.NO_ERROR);
    }

    /**
    * @notice Accepts new implementation of comptroller. msg.sender must be pendingImplementation
    * @dev Admin function for new implementation to accept it's role as implementation
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
    function _acceptImplementation() public returns (uint) {
        // Check caller is pendingImplementation and pendingImplementation ≠ address(0)
        if (msg.sender != pendingComptrollerImplementation || pendingComptrollerImplementation == add
            return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_PENDING_IMPLEMENTATION_ADDRESS_CHECK);
        }

        // Save current values for inclusion in log
        address oldImplementation = comptrollerImplementation;
        address oldPendingImplementation = pendingComptrollerImplementation;

        comptrollerImplementation = pendingComptrollerImplementation;

        pendingComptrollerImplementation = address(0);

        emit NewImplementation(oldImplementation, comptrollerImplementation);
        emit NewPendingImplementation(oldPendingImplementation, pendingComptrollerImplementation);

        return uint(Error.NO_ERROR);
    }


    /**
      * @notice Begins transfer of admin rights. The newPendingAdmin must call `_acceptAdmin` to fina
      * @dev Admin function to begin change of admin. The newPendingAdmin must call `_acceptAdmin` to
      * @param newPendingAdmin New pending admin.
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setPendingAdmin(address newPendingAdmin) public returns (uint) {
        // Check caller = admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PENDING_ADMIN_OWNER_CHECK);
        }

        // Save current value, if any, for inclusion in log
        address oldPendingAdmin = pendingAdmin;

        // Store pendingAdmin with value newPendingAdmin
        pendingAdmin = newPendingAdmin;

        // Emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin)
        emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
      * @dev Admin function for pending admin to accept role and update admin
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _acceptAdmin() public returns (uint) {
        // Check caller is pendingAdmin and pendingAdmin ≠ address(0)
        if (msg.sender != pendingAdmin || msg.sender == address(0)) {
            return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_ADMIN_PENDING_ADMIN_CHECK);
        }

        // Save current values for inclusion in log
        address oldAdmin = admin;
        address oldPendingAdmin = pendingAdmin;

        // Store admin with value pendingAdmin
        admin = pendingAdmin;

        // Clear the pending value
        pendingAdmin = address(0);

        emit NewAdmin(oldAdmin, admin);
        emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);

        return uint(Error.NO_ERROR);
    }

    /**
     * @dev Delegates execution to an implementation contract.
     * It returns to the external caller whatever the implementation returns
     * or forwards reverts.
     */
    function () payable external {
        // delegate all other functions to current implementation
        (bool success, ) = comptrollerImplementation.delegatecall(msg.data);
        assembly {
            let free_mem_ptr := mload(0x40)
            returndatacopy(free_mem_ptr, 0, returndatasize)

            switch success
            case 0 { revert(free_mem_ptr, returndatasize) }
            default { return(free_mem_ptr, returndatasize) }
        }
    }
}
```

Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Unexpected Blockchain Currency

- **Description:**
  Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain

Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Delegatecall

- **Description:**
The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whBlockchain Currency a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifiers can lead to some devestating vulernabilities in smart contracts as will be discussed in this section.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Entropy Illusion

- **Description:**
All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

External Contract Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send())

fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
  The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Denial Of Service (DOS)

- **Description:**
  This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Block Timestamp Manipulation

- **Description:**
  Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Constructors with Care

- **Description:**
  Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Unintialised Storage Pointers

- **Description:**
  The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

tx.origin Authentication

- **Description:**
  Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

armors.io

contact@armors.io