知道创宇

密级 ★ | 项目组公开

# 智能合约审计报告

安全状态

## 安全

★ ★ ★ ★ ★

主测人： 知道创宇区块链安全研究团队

# 版本说明

| 修订内容 | 时间 | 修订者 | 版本号 |
| --- | --- | --- | --- |
| 编写文档 | 20210121 | 知道创宇区块链安全研究团队 | V1.0 |

# 文档信息

| 文档名称 | 文档版本 | 报告编号 | 保密级别 |
| --- | --- | --- | --- |
| Box 智能合约审计报告 | V1.0 | 4a332ae36a764183acd8d082250f5103 | 项目组公开 |

# 声明

创宇仅就本报告出具前已经发生或存在的事实出具本报告，并就此承担相应责任。对于出具以后发生或存在的事实，创宇无法判断其智能合约安全状况，亦不对此承担责任。本报告所作的安全审计分析及其他内容，仅基于信息提供者截至本报告出具时向创宇提供的文件和资料。创宇假设:已提供资料不存在缺失、被篡改、删减或隐瞒的情形。如已提供资料信息缺失、被篡改、删减、隐瞒或反映的情况与实际情况不符的，创宇对由此而导致的损失和不利影响不承担任何责任。

# 目录

# 1. 综述

本次报告有效测试时间是从 2021 年 1 月 14 日开始到 2021 年 1 月 21 日结束，在此期间针对 **Box 智能合约代码**的安全性和规范性进行审计并以此作为报告统计依据。

此次测试中，知道创宇工程师对智能合约的常见漏洞（见第四章节）进行了全面的分析，同时对业务逻辑层面进行审计，未发现存在相关安全风险，综合评定为**通过**。

**本次智能合约安全审计结果：通过**

由于本次测试过程在非生产环境下进行，所有代码均为最新备份，测试过程均与相关接口人进行沟通，并在操作风险可控的情况下进行相关测试操作，以规避测试过程中的生产运营风险、代码安全风险。

**本次审计的报告信息：**

**报告编号：4a332ae36a764183acd8d082250f5103**

**报告查询地址链接:**

https://attest.im/attestation/searchresult?qurey=4a332ae36a764183acd8d082250f5103

**本次审计的目标信息：**

| 条目 | 描述 | |
|---|---|---|
| **Token 名称** | Box | |
| **代码类型** | 波场智能合约 | |
| **代码语言** | Solidity | |
| **合约地址** | Comptroller | TBfThHGPu78EapCnpLetyKBPZE2m48t2er |
| | AssertPriceOracle | TQ5wiYRF68N9AwgK13tnM1uES2ydkAinqs |

| | | |
|---|---|---|
| | JumpRateModel | THsxeUhV6qAZCPX4GejMW4KYNZbMxPRtn8 |
| | CNative | TKyotncVJxFxztu7DTv3zdmqdTUsYsFtFR |
| | CErc20Delegator | TJJpqmQVozm6KbxsshjrMJkHahZ3AHfCuB |
| | CErc20Delegator | TG6z24Wz7uy5Qi8Skq8p3hzAidGSqGQymd |
| | CErc20Delegator | TYfFQsuBYgbhG5WX5K73fbjcvFZM7f3Wsd |
| | CErc20Delegator | TMhtKRofPqV1UdtVnrexR5kUuoXAEzHwRX |
| | CErc20Delegator | TM8uyte1PR8knWsVtboFFmRcJZ8ActSgS4 |
| | CErc20Delegator | TLaEhK7UjEEr4Pf7DmXLNCX4adSbhn1tkw |

**合约文件及哈希：**

| 合约文件 | MD5 |
|---|---|
| **FeeManager.sol** | C5F6B2EFE2EC91101911B77EC6261D23 |
| **CarefulMath.sol** | 6E307FB2C177144BFABFD26C9DEF29E8 |
| **Context.sol** | EE8F37572F82A54CE72912BAD96343A1 |
| **Controller.sol** | F3847D2C3ED47FBB8EF47F7A5D0B4050 |
| **EnumerableSet.sol** | 728294CE4C12208919076D861B172F18 |
| **ERC20.sol** | FF18BF86C5A674DF6C4173E8AAB7566C |
| **ERC20Burnable.sol** | C4F5820744C45490CFD12B3A625E667C |
| **ERC20Detailed.sol** | A1A39233394F3C0A00CE348C61433FAA |
| **ERC20Mintable.sol** | D0828425DBB042C158558788C4BBC338 |
| **Exponential.sol** | 95B0C75925B3B337517ECCA57DB03AE5 |
| **IERC20.sol** | 183FF1F5E5F04BCFA85CEE597BCB0236 |
| **Math.sol** | 32FA90BA71DE78F2ADBE8A803D2D90B3 |
| **MinterRole.sol** | 2D9EC4DE161520693C92BD68EBADF363 |

| | |
|---|---|
| **NativeAddressLib.sol** | 648202C2B321BB5A7B8DD1B160E047E7 |
| **Ownable.sol** | 036CB43163C154F43DB99590E741E801 |
| **Roles.sol** | 118D44AA29245B854A8785A01AF0C4E9 |
| **SafeMath.sol** | 59D6A4434F7EAF874949A63A34C16B66 |
| **AssertPriceOracle.sol** | B0A22C349F87E062F611E4C86B939C13 |
| **IRewardPool.sol** | 4D853C13000099FFFEA53B555F7AC85E |
| **LPTokenWrapper.sol** | 0CE4424EB69028ED393E8535151E510F |
| **Reservoir.sol** | A4D53F03C465C07C113390F2B108AB6B |
| **RewardPool.sol** | 08EA94731298E0D95A20B7B95E5B876B |
| **RewardToken.sol** | 2A63E39CA8D584412CDE42301BD3A12B |
| **BaseJumpRateModelV2.sol** | 407955E406690C5430F328AE709ADB73 |
| **CErc20.sol** | BEFEDDD76F6DF03DC0CB03C685BB2E64 |
| **CErc20Delegate.sol** | BC0D3DF68E0518CD4321991010762BBB |
| **CErc20Delegator.sol** | 9A4603DE27C01987179D2EBEAD6EAD3E |
| **CErc20Immutable.sol** | A0B86D92E99899F710CCC322B25A536D |
| **CNative.sol** | 1A25DB4701976684F713C58D333EBE5E |
| **Comptroller.sol** | 0522C49F8B9858EF03BD9E1FF8A1DE7E |
| **CToken.sol** | AD93767AF10DCF651D9C516BD479D1D4 |
| **ErrorReporter.sol** | 66D35BE7A0050D7BE978A1E044966028 |
| **JumpRateModel.sol** | 4AA25F4B57B6044E2F203E582E5B44E5 |
| **JumpRateModelV2.sol** | B58D68DE521A209AC770027FF490A3C3 |

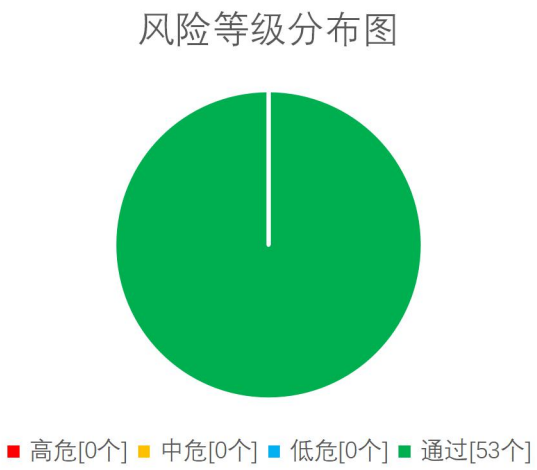
| | |
|---|---|
| **LegacyJumpRateModelV2.sol** | 430C924A87809CCB44927608E1857387 |
| **Maximillion.sol** | 037442A306408EF0BAC9B549E361C19A |
| **Migrations.sol** | AEB82FB35EFAC0EC173CD1376279CE67 |
| **RiskController.sol** | 8C34694F39172DD5CBBE7E056AA67F20 |
| **Unitroller.sol** | 9BC51EF9B26485B358E8330F194462D2 |
| **WhitePaperInterestRateModel.sol** | 10C844936BF1C848752D37FE13F36A40 |

# 2. 代码漏洞分析

## 2.1 漏洞等级分布

本次漏洞风险按等级统计:

| 安全风险等级个数统计表 | | | |
|---|---|---|---|
| 高危 | 中危 | 低危 | 通过 |
| 0 | 0 | 0 | 53 |



## 2.2 审计结果汇总说明

| 审计结果 | | | |
|---|---|---|---|
| 审计项目 | 审计内容 | 状态 | 描述 |
| 业务安全 | SwapToken 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | addTokenPoolLiquidity 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | SetBasePoolToken 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | SetUnderlyingOracle 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | GetUnderlyingPrice 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | Stake 质押逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | Withdraw 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | mintToken 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | 增加流动性池子逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | DistributionToken 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | ClaimFund 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | Exit 退出逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | NotifyRewardAmount 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | UpdateJumpRateModel 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | getBorrowRateInternal 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | Mint 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | Redeem 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | Borrow 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | repayBorrow 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | LiquidityBorrow 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | _addReserves 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | _reduceReserves 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |
| | isConstract 逻辑设计 | 通过 | 经检测，不存在该安全问题。 |

| | | | |
|---|---|---|---|
| 编码安全 | 重入攻击检测 | 通过 | 经检测，不存在该安全问题。 |
| | 重放攻击检测 | 通过 | 经检测，不存在该安全问题。 |
| | 重排攻击检测 | 通过 | 经检测，不存在该安全问题。 |
| | 数值溢出检测 | 通过 | 经检测，不存在该安全问题。 |
| | 算数精度误差 | 通过 | 经检测，不存在该安全问题。 |
| | 访问控制缺陷检测 | 通过 | 经检测，不存在该安全问题。 |
| | tx.origon 身份验证 | 通过 | 经检测，不存在该安全问题。 |
| | call 注入攻击 | 通过 | 经检测，不存在该安全问题。 |
| | 返回值调用验证 | 通过 | 经检测，不存在该安全问题。 |
| | 未初始化的存储指针 | 通过 | 经检测，不存在该安全问题。 |
| | 错误使用随机数检测 | 通过 | 经检测，不存在该安全问题。 |
| | 交易顺序依赖检测 | 通过 | 经检测，不存在该安全问题。 |
| | 拒绝服务攻击检测 | 通过 | 经检测，不存在该安全问题。 |
| | 假充值漏洞检测 | 通过 | 经检测，不存在该安全问题。 |
| | 增发代币漏洞检测 | 通过 | 经检测，不存在该安全问题。 |
| | 冻结账户绕过检测 | 通过 | 经检测，不存在该安全问题。 |
| | 编译器版本安全 | 通过 | 经检测，不存在该安全问题。 |
| | 不推荐的编码方式 | 通过 | 经检测，不存在该安全问题。 |
| | 冗余代码 | 通过 | 经检测，不存在该安全问题。 |
| | 安全算数库的使用 | 通过 | 经检测，不存在该安全问题。 |
| | require/assert 的使用 | 通过 | 经检测，不存在该安全问题。 |
| | energy 消耗检测 | 通过 | 经检测，不存在该安全问题。 |
| | fallback 函数安全 | 通过 | 经检测，不存在该安全问题。 |
| | owner 权限控制 | 通过 | 经检测，不存在该安全问题。 |
| | 低级函数安全 | 通过 | 经检测，不存在该安全问题。 |
| | 变量覆盖 | 通过 | 经检测，不存在该安全问题。 |

| | | | |
|---|---|---|---|
| | **时间戳依赖攻击** | 通过 | 经检测，不存在该安全问题。 |
| | **不安全的接口使用** | 通过 | 经检测，不存在该安全问题。 |

# 3. 业务安全性检测

## 3.1 swapToken 逻辑设计【通过】

对 swapToken 代币互换逻辑设计进行审计，检查是否有对参数进行检查以及逻辑设计的合理性、函数调用者的权限检查等。

**审计结果**: swapToken 函数有对参数合法性进行检查，并通过 onlyController 对函数调用者权限进行约束，相关逻辑设计未发现安全风险。

```
function swapToken(
        address poolAddress, address tokenIn, address tokenOut
  ) onlyController public returns (uint){
      require(factory != address(0), "factory=0");//knownsec//地址非 0 检查
      require(IFactory(factory).isBPool(poolAddress), "!pool");//knwonsec//pool 地址检查
      uint _tokenInAmount = IERC20(tokenIn).balanceOf(address(this));//knownsec//计算要兑换的 token 数量
    IERC20(tokenIn).approve(poolAddress, _tokenInAmount);//knownsec//授权转账操作
    (uint tokenAmountOut,)=IBool(poolAddress).swapExactAmountIn(tokenIn, _tokenInAmount, tokenOut, 0, MAX);//knownsec//兑换操作
        return tokenAmountOut;//knownsec//返回可兑换的期望 token 的数量
}
  modifier onlyController() {
            require(isController(msg.sender), "!controller");//knownsec//权限检查
            _;
  }
```

**安全建议：**无。

## 3.2 addTokenPoolLiquidity 逻辑设计【通过】

对 addTokenPoolLiquidity 业务逻辑进行审计，检查是否有对参数的合法性做检查以及逻辑设计的合理性，是否存在溢出风险等。

**审计结果：**addTokenPoolLiquidity 有对参数合法性检查，并使用 SafeMath 进行数值运算操作，不存在溢出风险，相关逻辑设计无误未发现安全风险。

```
function addTokenPoolLiquidity() external {
        require(IERC20(boxToken).balanceOf(msg.sender) >= dexTokenAmount, "not enough token");//knwonsec//token 数量检查
        uint _amount = IERC20(baseToken).balanceOf(address(this));
        uint256 _goverFund = _amount.mul(goverFundDivRate).div(1e18);//knownsec// 计算 goverFund 数量
        IERC20(baseToken).transfer(governaddr, _goverFund);//knownsec// 划转 _goverFund 数量的 token 到 governAddr

        uint _tokenInAmount = _amount.mul(burnRate).div(1e18);
        IERC20(baseToken).approve(basePool, _tokenInAmount);
        IBool(basePool).swapExactAmountIn(baseToken, _tokenInAmount, boxToken, 0, MAX);
        //knownsec//更新_amount
        _amount = _amount.sub(_goverFund);
        _amount = _amount.sub(_tokenInAmount);
        IERC20(baseToken).approve(basePool, _amount);
        IBool(basePool).joinswapExternAmountIn(baseToken, _amount, 0);
    }
```

**安全建议**：无

## 3.3 setBasePoolToken 逻辑设计【通过】

对 setBasePoolToken 业务逻辑进行审计，检查是否有有对参数的合法性做检

查、相关逻辑设计是否合理、权限设计是否合理等。

**审计结果：**setBasePoolToken 函数通过 onlyController 来对函数调用者身份进行检查，通过通过 require 对参数进行合法性检查，相关逻辑设计合理无误。

```
function setBasePoolToken(address _pool, address _token, address _baseToken) onlyController public {
        //knownsec//相关参数检查
        require(factory != address(0), "factory is 0");
        require(IFactory(factory).isBPool(_pool), "!pool");
        require(IBool(_pool).isBound(_token), "not bound");
        require(IBool(_pool).isBound(_baseToken), "not bound");
        require(setBasePoolCount < 2, "limit set base pool");
        setBasePoolCount = setBasePoolCount.add(1);
        basePool = _pool;
        boxToken = _token;
        baseToken = _baseToken;
    }
```

**安全建议：**无

## 3.4 setUnderlyingOracle 逻辑设计【通过】

对 setUnderlyingOracle 逻辑设计进行审计，检查权限设计是否合理、逻辑设计是否存在安全风险等。

**审计结果：**setUnderlyingOracle 通过修饰器 onlyController 对调用者身份做检查，相关逻辑设计无误。

```
function setUnderlyingOracle(address underlying, address oracle) onlyController external {
        ILinkOracle(oracle).latestAnswer();
        underlyingOracles[underlying] = oracle;
    }
    //knownsec//controller 在构造函数中被初始化
    constructor(address _controller, address _wtrxAddress) Controller() public {
            setController(_controller);
            wtrxAddress = _wtrxAddress;
```

```
    }
    //knownsec//setContrller 逻辑
    function setController(address controller) public onlyOwner {
                require(controller != address(0), "controller is empty");
                emit SetController(controller, _controller);
                _controller = controller;
     }
```

**安全建议：**无

## 3.5 getUnderlyingPrice 逻辑设计【通过】

对 getUnderlyingPrice 逻辑设计进行审计，检查是否有对参数进行检查、逻辑设计是否存在错误等。

**审计结果：**未发现存在相关安全风险。

```
    function getUnderlyingPrice(CToken cToken) external view returns (uint){
                address _underlying = address(CErc20(address(cToken)).underlying());
                if (justLps[_underlying] == true) {
                    return calJustLpPrice(_underlying);//knownsec//调用 calJustLpPrice
                }
                if (abeloLps[_underlying] == true) {
                    return calAbeloLpPrice(_underlying);//knownsec//调用 calAbeloLpPrice
                }
                address oracle = underlyingOracles[_underlying];
                require(oracle != address(0), "oracle is 0");//knownsec//地址非零判断
                return toUint(ILinkOracle(oracle).latestAnswer());
        }
        //knownsec//calJustLpPrice 逻辑设计与实现
    function calJustLpPrice(address _pairAddress) public view returns (uint){
            address _underlying = justLpPairs[_pairAddress];
            require(underlyingOracles[_underlying] != address(0), "oracle is 0");//knownsec//检查 underlying 是否存在
            uint _totalSupply = IJustPair(_pairAddress).totalSupply().div(10 **
```

```
IJustPair(_pairAddress).decimals());//knownsec//计算 totalsupply
            uint _totalUnderlying = ERC20Detailed(_underlying).balanceOf(_pairAddress);
            uint _totalTrx = _pairAddress.balance;
            uint _underlyingDecimal = ERC20Detailed(_underlying).decimals();
            uint _lpPerUnderlying = _totalUnderlying.div(_totalSupply).div(10 **
_underlyingDecimal);
            uint _lpPerTrx = _totalTrx.div(_totalSupply).div(10 ** wtrxDecimals);
            address oracle = underlyingOracles[_underlying];
            uint _underLyingPrice = uint(ILinkOracle(oracle).latestAnswer());
            address wtrxOracle = underlyingOracles[wtrxAddress];
            uint _trxPrice = uint(ILinkOracle(wtrxOracle).latestAnswer());
            return _lpPerUnderlying.mul(_underLyingPrice).add(_lpPerTrx.mul(_trxPrice));
          }
         //knownsec//calAbeloLpPrice 逻辑设计与实现
        function calAbeloLpPrice(address _lpAddress) public view returns (uint){
            address _poolView = abeloLpViews[_lpAddress];
            uint total;
            address[] memory tokens = IPoolView(_poolView).getCurrentTokens();
           for (uint i = 0; i < 2; i++) {
               address token = tokens[i];
                uint _lpPerToken = IPoolView(_poolView).getTokenAmountPerLp(token);
                address oracle = underlyingOracles[token];
                uint _tokenPrice = uint(ILinkOracle(oracle).latestAnswer());
                total = total.add(_lpPerToken.mul(_tokenPrice));
            }
             return total;
        }
```

**安全建议**：无

## 3.6 stake 质押逻辑设计【通过】

对 stake 质押逻辑进行审计，检查逻辑设计的合理性以及是否存在溢出风险。

**审计结果：**stake 采用了 SafeMath 进行数值运算操作较为安全，逻辑设计合

理，未发现相关安全风险。

```
function stake(uint256 amount) public {
        _totalSupply = _totalSupply.add(amount);
        _balances[msg.sender] = _balances[msg.sender].add(amount);
        lpToken.transferFrom(msg.sender, address(this), amount);
}
```

**安全建议：**无

## 3.7 withdraw 逻辑设计【通过】

对 withdraw 提款逻辑审计，检查逻辑设计的合理性以及是否存在溢出风险。

**审计结果：**withdraw 采用了 SafeMath 进行数值运算操作较为安全，逻辑设计合理，未发现相关安全风险

```
    function withdraw(uint256 amount) public {
        _totalSupply = _totalSupply.sub(amount);
        _balances[msg.sender] = _balances[msg.sender].sub(amount);
        lpToken.transfer(msg.sender, amount);
    }
```

**安全建议：**无

## 3.8 mintToken 逻辑设计【通过】

对 mintToken 逻辑进行审计，检查逻辑设计的合理性以及是否存在溢出风险，权限设计是否合理等。

**审计结果：**mintToken 由修饰器 onlyOwner 修饰，只能由合约的 Owner 调用且 mintToken 只能铸币一次，不能无限增发，同时在铸币过程中采用 SafeMath 函数进行数值运算操作，不存在溢出风险。

```
function mintToken() onlyOwner public {
        rewardToken.mint(address(this));//knownsec//调用 mint 函数来实现铸币
}
function mint(address _to) public onlyOwner {
        require(totalSupply() == 0, "mint once");//knownsec//只能铸币一次
        _mint(_to, _totalSupply);
```

```
    }
    function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address");
        _totalSupply = _totalSupply.add(amount);//knownsec//增加总量
        _balances[account] = _balances[account].add(amount);//knownsec//增加 token
        emit Transfer(address(0), account, amount);
    }
```

**安全建议：**无

## 3.9 增加流动性池子逻辑设计【通过】

对增加流动性池子逻辑进行审计，检查是否有对参数进行检查，权限设计是否合理，逻辑设计是否合理等。

**审计结果：**经测试，发现有对权限进行检查，且在增加流动性池子时有对当前池子是否已经存在进行判断，相关逻辑设计无误。

```
    function add(
        address _contractAddress,
        uint256 _allocPoint
    ) public onlyController adjustProduct {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            PoolInfo memory pool = poolInfo[pid];
            require(pool.contractAddress != _contractAddress, "contract is exist");//knownsec//判断当前流动性池子是否已经存在
        }
        uint256 lastRewardBlock = block.number > startBlock
        ? block.number
        : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);//knownsec//更新
        poolInfo.push(//knownsec//增加新的 LP 逻辑
            PoolInfo(
            {
```

```
                allocPoint : _allocPoint,
                contractAddress : _contractAddress
                }
                )
            );
    }
    modifier onlyController() {
        require(isController(msg.sender), "!controller");//knownsec//权限检查
            _;
    }
    function isController(address account) public view returns (bool) {
            return account == _controller;
    }
    modifier adjustProduct()    {//knownsec//调整处理
            _;
            if (block.number >= periodEndBlock) {
                if (tokenPerBlock > MIN_TOKEN_REWARD) {
                    tokenPerBlock = tokenPerBlock.mul(90).div(100);
                }
                if (tokenPerBlock < MIN_TOKEN_REWARD) {
                    tokenPerBlock = MIN_TOKEN_REWARD;
                }
                periodEndBlock = block.number.add(duration);//更新周期结束块
            }}
```

**安全建议：**无。

## 3.10 distributionToken 逻辑设计【通过】

对 distributionToken 逻辑进行审计，检查是否有对参数进行校验以及逻辑设计是否合理等。

**审计结果**：未发现相关安全风险。

```
    function distributionToken() adjustProduct public {
```

```
        if (block.number < startBlock) {//knownsec//判断当前是否处于一个周期内
            return;
        }
        if (poolLastRewardBlock >= periodEndBlock) {//knownsec//判断是否结束
            return;
        }
        uint multiplier = 1;
        if (periodEndBlock > block.number) {
            multiplier = periodEndBlock - block.number;//knownsec//计算到一个周期结束还剩余多少个区块
        }
        uint _reward = multiplier.mul(tokenPerBlock);
      //knownsec//按比例分成
        uint goverFund = calRate(_reward, goverFundDivRate);
        uint insuranceFund = calRate(_reward, insuranceFundDivRate);
        uint teamFund = calRate(_reward, teamFundDivRate);
        uint liquidityFund = calRate(_reward, liquidityRate);

        uint distriReward = _reward.sub(goverFund.add(insuranceFund).add(teamFund).add(liquidityFund));

        safeTokenTransfer(address(comptroller), distriReward);
        Comptroller(comptroller)._setCompRate(calRate(tokenPerBlock, comptrollerRate));
        //knownsec//池子分成
        uint length = poolInfo.length;
        for (uint pid = 0; pid < length; ++pid) {
            PoolInfo storage pool = poolInfo[pid];
            uint256 poolReward = liquidityFund.mul(pool.allocPoint).div(totalAllocPoint);
            safeTokenTransfer(pool.contractAddress, poolReward);
            IRewardPool(pool.contractAddress).notifyRewardAmount(poolReward);
        }
```

```
            poolLastRewardBlock = periodEndBlock;
        }
        // Safe pickle transfer function, just in case if rounding error causes pool to not have enough dex.
        //knownsec//转账逻辑
        function safeTokenTransfer(address _to, uint256 _amount) internal {
            uint256 pickleBal = rewardToken.balanceOf(address(this));
            if (_amount > pickleBal) {
                rewardToken.transfer(_to, pickleBal);
            } else {
                rewardToken.transfer(_to, _amount);
            }
        }
```

**安全建议**：无

## 3.11 claimFund 逻辑设计【通过】

对 claimFund 逻辑进行审计，检查是否有对参数进行校验以及逻辑设计是否合理等。

**审计结果：**claimFund 有对参数进行合法性检查，相关逻辑设计无误。

```
    // Distribute tokens according to teh speed of the block
    function claimFund() onlyController external {
        if (block.number <= fundLastRewardBlock) {//knownsec//检查是否到最后一个区块
            return;
        }
        uint256 multiplier = block.number - fundLastRewardBlock;
        uint256 boxReward = multiplier.mul(tokenPerBlock);//knownsec//计算奖励
        uint goverFund = calRate(boxReward, goverFundDivRate);
        uint insuranceFund = calRate(boxReward, insuranceFundDivRate);
        uint teamFund = calRate(boxReward, teamFundDivRate);
        rewardToken.transfer(governaddr, goverFund);
```

```
            rewardToken.transfer(insuranceaddr, insuranceFund);
            uint256 perDevFund = teamFund.div(teamAddrs.length());//knownsec//team
分成
            for (uint256 i = 0; i < teamAddrs.length() - 1; i++) {
                rewardToken.transfer(teamAddrs.get(i), perDevFund);
            }
            uint256 remainFund = teamFund.sub(perDevFund.mul(teamAddrs.length() -
1));
            rewardToken.transfer(teamAddrs.get(teamAddrs.length() - 1), remainFund);
            goverFund = 0;
            insuranceFund = 0;
            teamFund = 0;
            fundLastRewardBlock = block.number;
            distributionToken();
        }
```

**安全建议**：无

## 3.12 exit 退出逻辑设计【通过】

对 exit 退出逻辑进行审计，检查逻辑设计是否合理，参数是否有检查等。

**审计结果：**未发现相关安全风险。

```
function exit() external {
        withdraw(balanceOf(msg.sender));//knownsec//提款逻辑
        getReward();//knownsec//提取奖励
}
 function withdraw(uint256 amount) public updateReward(msg.sender) {
        require(amount > 0, "Cannot withdraw 0");//knownsec//资产数量检查
        uint exitAmount = amount.mul(exitFee).div(1e18);//knownsec//exitfee 计算
        uint _amount = amount.sub(exitAmount);//knownsec//可提取资产
        super.withdraw(_amount);//knownsec//提取资产
        emit Withdrawn(msg.sender, amount);
}
  //knownsec//计算奖励
```

```
    function getReward() public updateReward(msg.sender) {
        uint256 reward = earned(msg.sender);
         if (reward > 0) {//knownsec//检查 reward 是否大于 0
            rewards[msg.sender] = 0;
            // If it is a normal user and not smart contract,
            // then the requirement will pass
            // If it is a smart contract, then
            // make sure that it is not on our greyList.
            if (tx.origin == msg.sender) {//knownsec//调用者身份检查
                rewardToken.transfer(msg.sender, reward);//knownsec//提取
                emit RewardPaid(msg.sender, reward);
            } else {
                emit RewardDenied(msg.sender, reward);
            }
        }
    }
    //knownsec//收益计算
     function earned(address account) public view returns (uint256) {
        return
        balanceOf(account)
        .mul(rewardPerToken().sub(userRewardPerTokenPaid[account]))
        .div(1e18)
        .add(rewards[account]);
    }
```

**安全建议**：无

## 3.13 notifyRewardAmount 逻辑设计【通过】

对 notifyRewardAmount 逻辑进行审计，检查是否有对参数的合法性进行检查、逻辑设计是否合理、权限设计是否合理等。

**审计结果：**notifyRewardAmount 有对调用者身份进行检查同时有对参数进行检查，函数逻辑设计无误。

```
function notifyRewardAmount(uint256 reward)
```

```
        external
        updateReward(address(0))
        {
            require(msg.sender == controller() || msg.sender == reservoirAddress, "only controller and reservoir");//knownsec//权限检查

            require(reward < uint(- 1) / 1e18, "the notified reward cannot invoke multiplication overflow");//knownsec//溢出检查

            if (block.number >= periodFinish) {
                rewardRate = reward.div(duration);
            } else {
                uint256 remaining = periodFinish.sub(block.number);
                uint256 leftover = remaining.mul(rewardRate);
                rewardRate = reward.add(leftover).div(duration);
            }
            lastUpdateTime = block.number;//knownsec//更新 lastUPdateTime
            periodFinish = block.number.add(duration);//knownsec//更新周期
            emit RewardAdded(reward);
        }
```

**安全建议**：无

## 3.14 updateJumpRateModel 逻辑设计【通过】

对 updateJumpRateModel 更新利率模型参数函数进行审计，检查是否有对调用者身份进行权限检查、逻辑设计是否存在安全风险等。

**审计结果：**updateJumpRateModel 有对调用者身份进行检查

```
    function updateJumpRateModel(
            uint baseRatePerYear,
            uint multiplierPerYear,
            uint jumpMultiplierPerYear,
            uint kink_
```

```
        ) external {
            require(msg.sender == owner, "only the owner may call this function.");//knownsec//对调用者身份进行权限检查
            updateJumpRateModelInternal(baseRatePerYear, multiplierPerYear, jumpMultiplierPerYear, kink_);
        }
    //knownsec//更新利率模型参数
    function updateJumpRateModelInternal(
        uint baseRatePerYear,
        uint multiplierPerYear,
        uint jumpMultiplierPerYear,
        uint kink_
    ) internal {
        baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
        multiplierPerBlock = (multiplierPerYear.mul(1e18)).div(blocksPerYear.mul(kink_));
        jumpMultiplierPerBlock = jumpMultiplierPerYear.div(blocksPerYear);
        kink = kink_;
        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock, jumpMultiplierPerBlock, kink);
    }
```

**安全建议**：无

## 3.15 getBorrowRateInternal 逻辑设计【通过】

对 getBorrowRateInternal 借款利率逻辑进行审计，检查是否有对参数进行校验等。

**审计结果：**未发现相关安全风险。

```
    function getBorrowRateInternal(uint cash, uint borrows, uint reserves) internal view returns (uint) {
        uint util = utilizationRate(cash, borrows, reserves);
        if (util <= kink) {
            return util.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
```

```
        } else {
            uint normalRate = kink.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
            uint excessUtil = util.sub(kink);
            return excessUtil.mul(jumpMultiplierPerBlock).div(1e18).add(normalRate);
        }
    }
    //knownsec//计算市场的利率
    function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }
        return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
    }
```

**安全建议：**无

## 3.16 getSupplyRate 逻辑设计【通过】

对 getSupplyRate 逻辑进行审计，检查是否存在溢出风险以及逻辑设计错误等。

**审计结果：**getSupplyRate 使用 SafeMath 进行数值运算操作，不存在溢出风险，相关逻辑设计无误。

```
function getSupplyRate(
        uint cash,
        uint borrows,
        uint reserves,
        uint reserveFactorMantissa
    ) public view returns (uint) {
        uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
        uint borrowRate = getBorrowRateInternal(cash, borrows, reserves);
```

```
            uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
            return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
}
    //knownsec//借款利率
    function getBorrowRateInternal(uint cash, uint borrows, uint reserves) internal view returns (uint) {
            uint util = utilizationRate(cash, borrows, reserves);

            if (util <= kink) {
                return util.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
            } else {
                uint normalRate = kink.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
                uint excessUtil = util.sub(kink);
                return excessUtil.mul(jumpMultiplierPerBlock).div(1e18).add(normalRate);
            }
    }
```

**安全建议：**无

## 3.17 mint 逻辑设计【通过】

对 mint 逻辑设计进行审计，检查是否有对参数进行校验、是否有对函数调用者权限进行检查、逻辑设计是否存在错误等。

**审计结果：**mint 铸币函数逻辑设计无误，有明确的权限检查。

```
    function mint(uint mintAmount) external returns (uint) {
            (uint err,) = mintInternal(mintAmount);
            return err;
    }
    function mintInternal(uint mintAmount) internal nonReentrant returns (uint, uint) {
            address riskControl = Comptroller(address(comptroller)).getRiskControl();
            if (riskControl != address(0)) {//knownsec//非空检查
                uint risk = RiskController(riskControl).checkMintRisk(address(this), msg.sender);
```

```
            require(risk == 0, 'risk control');//knownsec//risk 风险检查
        }
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {//knownsec//错误类型检查
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow failed
            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 to
        return mintFresh(msg.sender, mintAmount);

    }
    //knownsec//mintFresh 逻辑设计
    function mintFresh(address minter, uint mintAmount) internal returns (uint, uint) {
        /* Fail if mint not allowed */
        uint allowed = comptroller.mintAllowed(address(this), minter, mintAmount);//knownsec//权限检查
        if (allowed != 0) {
            return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.MINT_COMPTROLLER_REJECTION, allowed), 0);
        }

        /* 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(vars.mathErr)), 0);
            }

            //////////////////////////
            // 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 : vars.exchangeRateMantissa})
            );
            require(vars.mathErr == MathError.NO_ERROR, "MINT_EXCHANGE_CALCULATION_FAILED");
```

```
        /*
         * We calculate the new total supply of cTokens and minter token balance, checking for overflow:
         *    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);
    }
```

**安全建议**：无

## 3.18 redeem 逻辑设计【通过】

对 redeem 逻辑设计进行审计，检查逻辑设计是否合理，参数是否有做检查等。

**审计结果：**未发现相关安全风险。

```
function redeem(uint redeemTokens) external returns (uint) {
        return redeemInternal(redeemTokens);
    }
    function redeemInternal(uint redeemTokens) internal nonReentrant returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {//knownsec//错误检查
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted redeem failed
            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);
    }
    //knownsec//将应计利息应用于借款和准备金总额,这将计算从最后一个检查点块到当前块的累计利息，并将新的检查点写入存储器
    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, reservesPrior);
        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_CALCULATION_FAILED,
                    uint(mathErr)
                );
            }

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

FailureInfo.ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED,
                    uint(mathErr)
                );
            }

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

FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED,
                    uint(mathErr)
                );
            }

            (mathErr, totalReservesNew) = mulScalarTruncateAddUInt(
                Exp({mantissa : reserveFactorMantissa}),
```

```
                interestAccumulated,
                reservesPrior
            );
            if (mathErr != MathError.NO_ERROR) {
                return failOpaque(
                    Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED,
                    uint(mathErr)
                );
            }

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

            ////////////////////////
            // 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);
    }
  //knownsec//redeemFresh 逻辑设计
    function redeemFresh(address payable redeemer, uint redeemTokensIn, uint redeemAmountIn) internal returns (uint) {
        require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of redeemTokensIn or redeemAmountIn must be zero");
        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(vars.mathErr));
        }
        /* 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.exchangeRateMantissa}), redeemTokensIn
            );
            if (vars.mathErr != MathError.NO_ERROR) {
                return failOpaque(
```

```
                    Error.MATH_ERROR,

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

FailureInfo.REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED,
                    uint(vars.mathErr)
                );
            }
            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, allowed);
        }
```

```
        /* 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_FAILED,
                uint(vars.mathErr)
            );
        }

        (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_FAILED,
                uint(vars.mathErr)
            );
        }
```

```
        /* 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 occurred.
         */
        uint _amount = vars.redeemAmount;
        uint _feeAmount = div_(mul_(_amount, redeemFee), expScale);
        totalReserves = add_(totalReserves, _feeAmount);
        _amount = sub_(_amount, _feeAmount);
        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);
    }
```

**安全建议：**无

## 3.19 borrow 逻辑设计【通过】

对 borrow 资产借出逻辑设计进行审计，检查是否有对参数进行合法性校验、逻辑设计是否合理等。

**审计结果**：borrow 逻辑设计无误，未发现存在相关安全风险。

```
    function borrow(uint borrowAmount) external returns (uint) {
        return borrowInternal(borrowAmount);
    }
    function borrowInternal(uint borrowAmount) internal nonReentrant returns (uint) {
        address riskControl = Comptroller(address(comptroller)).getRiskControl();
        if (riskControl != address(0)) {//knownsec//地址检查
            uint risk = RiskController(riskControl).checkMintRisk(address(this), msg.sender);
            require(risk == 0, 'risk control');//knownsec//风险检查
        }
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow failed
            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);
    }
```

```
    //knownsec//borrowFresh 逻辑设计
    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, allowed);
        }

        /* 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_FAILED,
                    uint(vars.mathErr)
                );
            }

            (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_CALCULATION_FAIL
ED,
                    uint(vars.mathErr)
                );
            }

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

FailureInfo.BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED,
                    uint(vars.mathErr)
                );
            }

            ////////////////////////
            // 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 occurred.
             */
            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);
        }
```

**安全建议：**无

## 3.20 repayBorrow 逻辑设计【通过】

对 repayBorrow 还款逻辑设计进行审计，检查逻辑设计的合理性以及是否有对参数进行检查等。

**审计结果：**未发现存在相关安全风险。

```
    function repayBorrow(uint repayAmount) external returns (uint) {
        (uint err,) = repayBorrowInternal(repayAmount);
```

```
        return err;
    }
    function repayBorrowInternal(uint repayAmount) internal nonReentrant returns (uint, uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {//knownsec//错误检查
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow failed
            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);
    }
    //knownsec//repayBorrowFresh 逻辑设计
    function repayBorrowFresh(address payer, address borrower, uint repayAmount) internal returns (uint, uint) {
        /* 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_REJECTION,
                allowed
            ), 0);
        }
        /* 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_CALCULATION_FAILED
,
                uint(vars.mathErr)
            ), 0);
        }

        /* 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 occurred.
            *  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_CALCULATION_FAILED");

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

            /* 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.totalBorrowsNew);
            /* We call the defense hook */
            comptroller.repayBorrowVerify(address(this), payer, borrower, vars.actualRepayAmount, vars.borrowerIndex);
```

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

**安全建议：**无

## 3.21 repayBorrowBehalf 逻辑设计【通过】

对 repayBorrowBehalf 归还一半逻辑进行审计，检查是否有对参数进行检查以及逻辑设计的合理性等。

**审计结果**：未发现存在相关安全风险。

```
    function repayBorrowBehalf(address borrower, uint repayAmount) external returns (uint)
{
        (uint err,) = repayBorrowBehalfInternal(borrower, repayAmount);
        return err;
}
    function repayBorrowBehalfInternal(address borrower, 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
attempted borrow failed
            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);
    }
```

**安全建议：**无

## 3.22 liquidateBorrow 逻辑设计【通过】

对 liquidateBorrow 逻辑设计进行审计，检查逻辑设计的合理性以及参数检查

等。

**审计结果：**未发现存在相关安全风险。

```
    function liquidateBorrow(
        address borrower,
        uint repayAmount,
        CTokenInterface cTokenCollateral
    ) external returns (uint) {
        (uint err,) = liquidateBorrowInternal(borrower, repayAmount, cTokenCollateral);
        return err;
    }
    //knownsec//liquidateBorrowInternal 逻辑
    function liquidateBorrowInternal(
        address borrower,
        uint repayAmount,
        CTokenInterface cTokenCollateral
    ) 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 attempted liquidation failed
            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 attempted liquidation failed
            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);
    }
    //knownsec//liquidateBorrowFresh 逻辑
    function liquidateBorrowFresh(
        address liquidator,
        address borrower,
        uint repayAmount,
        CTokenInterface cTokenCollateral
    ) internal returns (uint, uint) {
        /* Fail if liquidate not allowed */
        uint allowed = comptroller.liquidateBorrowAllowed(
            address(this),
            address(cTokenCollateral),
            liquidator,
            borrower,
            repayAmount
        );
        if (allowed != 0) {
            return (failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.LIQUIDATE_COMPTROLLER_REJECTION, allowed), 0);
        }
        /* 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_ZERO), 0);
        }

        /* Fail if repayAmount = -1 */
        if (repayAmount == uint(- 1)) {
            return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED, FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX), 0);
        }

        /* Fail if repayBorrow fails */
        (uint repayBorrowError, uint actualRepayAmount) = repayBorrowFresh(liquidator, borrower, repayAmount);
        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(this),
                address(cTokenCollateral),
                actualRepayAmount
            );
            require(amountSeizeError == uint(Error.NO_ERROR),
"LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED");
            /* 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 external call
            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), seizeTokens);

            /* We call the defense hook */
            comptroller.liquidateBorrowVerify(
                address(this),
                address(cTokenCollateral),
                liquidator,
                borrower,
                actualRepayAmount,
                seizeTokens
```

```
        );

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

**安全建议：**无

## 3.23 _addReserves 逻辑设计【通过】

对_addReserves 增加储备金逻辑进行审计，检查是否有对参数进行校验以及逻辑设计的合理性等。

**审计结果**：未发现存在相关安全风险。

```
    function _addReserves(uint addAmount) external returns (uint) {
        return _addReservesInternal(addAmount);
    }
    function _addReservesInternal(uint addAmount) internal nonReentrant returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {//knownsec//错误类型检查
            // accrueInterest emits logs on errors,
            // but on top of that we want to log the fact that an attempted reduce reserves
failed.
            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);//knownsec//调用 addReservesFresh
        return error;
    }
    //knownsec//addReservesFresh 逻辑
    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), actualAddAmount);
        }

        ////////////////////////
        // 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 occurred.
         *  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);
    }
```

**安全建议：**无

## 3.24 _reduceReserves 逻辑设计【通过】

对_reduceReserves 减少储备金逻辑进行审计，检查逻辑设计的合理性、权限设计与检查是否合理等。

**审计结果**：未发现存在相关安全风险。

```
    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 an attempted reduce reserves failed.
            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);
    }
    function _reduceReservesFresh(uint reduceAmount) internal returns (uint) {
        // totalReserves - reduceAmount
        uint totalReservesNew;

        // Check caller is admin
        if (msg.sender != admin) {//knownsec//权限检查
            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 occurred.
        doTransferOut(feeManager, reduceAmount);
        emit ReservesReduced(admin, reduceAmount, totalReservesNew);
        return uint(Error.NO_ERROR);
    }
```

**安全建议**：无

## 3.25 isContractCall 逻辑设计【通过】

对 isContractCall 合约检测逻辑进行审计，检查逻辑设计的合理性。

**审计结果：**未发现存在相关安全风险。

```
  function isContractCall() public view returns (bool isContract){
      return msg.sender != tx.origin;
  }
```

**安全建议**：无

# 4. 代码基本漏洞检测

## 4.1 重入攻击检测【通过】

重入漏洞是最著名的智能合约漏洞。

在 Solidity 中，调用其他地址的函数或者给合约地址转账，都会把自己的地址作为被调合约的 msg.sender 传递过去。此时，如果传递的 energy 足够的话，就可能会被对方进行重入攻击，在波场中对合约地址调用 transfer/sender，只会传递 2300 的 Energy，这不足以发起一次重入攻击。所以要一定避免通过 call.value 的方式，调用未知的合约地址。因为 call.value 可以传入远大于 2300 的 energy。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.2 重放攻击检测【通过】

合约中如果涉及委托管理的需求，应注意验证的不可复用性，避免重放攻击

在资产管理体系中，常有委托管理的情况，委托人将资产给受托人管理，委托人支付一定的费用给受托人。这个业务场景在智能合约中也比较普遍。。

**检测结果：**经检测，智能合约未使用 call 函数，不存在此漏洞。

**安全建议：**无。

## 4.3 重排攻击检测【通过】

重排攻击是指矿工或其他方试图通过将自己的信息插入列表(list)或映射

(mapping)中来与智能合约参与者进行“竞争”，从而使攻击者有机会将自己的信息存储到合约中。

**检测结果**:经检测，智能合约代码中不存在相关漏洞。

**安全建议**:无。

## 4.4 数值溢出检测【通过】

智能合约中的算数问题是指整数溢出和整数下溢。

Solidity 最多能处理 256 位的数字（2^256-1），最大数字增加 1 会溢出得到 0。同样，当数字为无符号类型时，0 减去 1 会下溢得到最大数字值。

整数溢出和下溢不是一种新类型的漏洞，但它们在智能合约中尤其危险。溢出情况会导致不正确的结果，特别是如果可能性未被预期，可能会影响程序的可靠性和安全性。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.5 算术精度误差【通过】

Solidity 作为一门编程语言具备和普通编程语言相似的数据结构设计，比如：变量、常量、函数、数组、函数、结构体等等，Solidity 和普通编程语言也有一个较大的区别——Solidity 没有浮点型，且 Solidity 所有的数值运算结果都只会是整数，不会出现小数的情况，同时也不允许定义小数类型数据。合约中的数值运算必不可少，而数值运算的设计有可能造成相对误差，例如同级运算：5/2*10=20，而 5*10/2=25，从而产生误差，在数据更大时产生的误差也会更大，更明显。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.6 访问控制检测【通过】

合约中不同函数应设置合理的权限

检查合约中各函数是否正确使用了 public、private 等关键词进行可见性修饰，检查合约是否正确定义并使用了 modifier 对关键函数进行访问限制，避免越权导致的问题。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.7 tx.origin 身份验证【通过】

tx.origin 是 Solidity 的一个全局变量，它遍历整个调用栈并返回最初发送调用（或事务）的帐户的地址。在智能合约中不恰当的使用此变量进行身份验证会使合约容易受到类似网络钓鱼的攻击。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.8 call 注入攻击【通过】

call 函数调用时，应该做严格的权限控制，或直接写死 call 调用的函数。

**检测结果：**经检测，智能合约未使用 call 函数，不存在此漏洞。

**安全建议：**无。

## 4.9 返回值调用验证【通过】

此问题多出现在和转币相关的智能合约中，故又称作静默失败发送或未经检查发送。

在 Solidity 中存在 transfer()、send()、call.value()等转币方法，都可以用于向某一地址发送 trx，其区别在于： transfer 发送失败时会 throw，并且进行状态回滚；只会传递 2300energy 供调用，防止重入攻击；send 发送失败时会返回 false；只会传递 2300energy 供调用，防止重入攻击；call.value 发送失败时会返回 false；传递所有可用 energy 进行调用（可通过传入 feeLimit 参数进行限制），不能有效防止重入攻击。

如果在代码中没有检查以上 send 和 call.value 转币函数的返回值，合约会继续执行后面的代码，可能由于 trx 发送失败而导致意外的结果。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.10 未初始化的储存指针【通过】

在 solidity 中允许一个特殊的数据结构为 struct 结构体，而函数内的局部变量默认使用 storage 或 memory 储存。

而存在 storage(存储器)和 memory(内存)是两个不同的概念，solidity 允许指针指向一个未初始化的引用，而未初始化的局部 stroage 会导致变量指向其他储存变量，导致变量覆盖，甚至其他更严重的后果，在开发中应该避免在函数中初始化 struct 变量。

**检测结果：**经检测，智能合约代码不使用结构体，不存在该问题。

**安全建议：**无。

## 4.11 错误使用随机数【通过】

智能合约中可能需要使用随机数，虽然 Solidity 提供的函数和变量可以访问明显难以预测的值，如 block.number 和 block.timestamp，但是它们通常或者比看起来更公开，或者受到矿工的影响，即这些随机数在一定程度上是可预测的，所以恶意用户通常可以复制它并依靠其不可预知性来攻击该功能。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.12 交易顺序依赖【通过】

由于矿工总是通过代表外部拥有地址（EOA）的代码获取 energy 费用，因此用户可以指定更高的费用以便更快地开展交易。由于波场区块链是公开的，每个人都可以看到其他人未决交易的内容。这意味着，如果某个用户提交了一个有价值的解决方案，恶意用户可以窃取该解决方案并以较高的费用复制其交易，以抢占原始解决方案。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.13 拒绝服务攻击【通过】

在波场的世界中，拒绝服务是致命的，遭受该类型攻击的智能合约可能永远

无法恢复正常工作状态。导致智能合约拒绝服务的原因可能有很多种，包括在作为交易接收方时的恶意行为，人为增加计算功能所需 energy 导致 energy 耗尽，滥用访问控制访问智能合约的 private 组件，利用混淆和疏忽等等。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.14 假充值漏洞【通过】

在代币合约的 transfer 函数对转账发起人(msg.sender)的余额检查用的是 if 判断方式，当 balances[msg.sender] < value 时进入 else 逻辑部分并 return false，最终没有抛出异常，我们认为仅 if/else 这种温和的判断方式在 transfer 这类敏感函数场景中是一种不严谨的编码方式。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.15 增发代币漏洞【通过】

检查在初始化代币总量后，代币合约中是否存在可能使代币总量增加的函数。

**检测结果：**经检测，智能合约代码中存在代币增发逻辑，但属于流动性池子铸币/销毁逻辑的正常业务需要且有明确的权限检查，遂评定为通过。

**安全建议：**无。

## 4.16 冻结账户绕过【通过】

检查代币合约中在转移代币时，是否存在未校验代币来源账户、发起账户、

目标账户是否被冻结的操作。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.17 编译器版本安全【通过】

检查合约代码实现中是否使用了安全的编译器版本

**检测结果：**经检测，智能合约代码中制定了编译器版本 0.5.8 以上，不存在该安全问题。

**安全建议：**无。

## 4.18 不推荐的编码方式【通过】

检查合约代码实现中是否有官方不推荐或弃用的编码方式

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.19 冗余代码【通过】

检查合约代码实现中是否包含冗余代码

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.20 安全算数库的使用【通过】

检查合约代码实现中是否使用了 SafeMath 安全算数库

**检测结果：**经检测，智能合约代码中已使用 SafeMath 安全算数库，不存在该安全问题。

**安全建议：**无。

## 4.21 require/assert 的使用【通过】

检查合约代码实现中 require 和 assert 语句使用的合理性

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.22 energy 消耗检测【通过】

检查 energy 的消耗是否超过区块最大限制

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.23 fallback 函数安全【通过】

检查合约代码实现中是否正确使用 fallback 函数

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.24 owner 权限控制【通过】

检查合约代码实现中的 owner 是否具有过高的权限。例如，任意修改其他账户余额等。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.25 低级函数安全【通过】

检查合约代码实现中低级函数（call/delegatecall）的使用是否存在安全漏洞

call 函数的执行上下文是在被调用的合约中；而 delegatecall 函数的执行上下文是在当前调用该函数的合约中

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.26 变量覆盖【通过】

检查合约代码实现中是否存在变量覆盖导致的安全问题

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.27 时间戳依赖攻击【通过】

数据块的时间戳通常来说都是使用矿工的本地时间，而这个时间大约能有 900 秒的范围波动，当其他节点接受一个新区块时，只需要验证时间戳是否晚于之前的区块并且与本地时间误差在 900 秒以内。一个矿工可以通过设置区块的时间戳来尽可能满足有利于他的条件来从中获利。

检查合约代码实现中是否存在有依赖于时间戳的关键功能

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.28 不安全的接口使用【通过】

检查合约代码实现中是否使用了不安全的接口

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。



# 5. 附录 A：合约代码

本次测试代码来源：

***FeeManager.sol***

```
pragma solidity ^0.5.9;

import "../token/RewardToken.sol";
import "../lib/Controller.sol";
import "../interface/IBool.sol";
```

```
import "../interface/IFactory.sol";
import "../interface/Converter.sol";
import "../lib/SafeMath.sol";

contract FeeManager is Controller {

    using SafeMath for uint;
    uint public constant MAX = 2 ** 256 - 1;
    uint public constant MIN_FEE = 1e12;
    uint public constant MAX_FEE = 1e18;

    uint public goverFundDivRate = 3 * 1e16;
    address public governaddr;

    uint public burnRate = 15 * 1e16;

    uint public dexTokenAmount = 10000 * 1e18;

    address public boxToken;
    address public baseToken;
    address public basePool;

    address public factory;

    uint private setFactoryManagerCount = 0;
    uint private setBasePoolCount = 0;
    address payable public wtrxAddress;

    mapping(address => address) public converters;


    constructor() Controller() public {
    }

    function() payable external {}


    function convertWtrx() external {
        uint _amount = address(this).balance;
        wtrxAddress.transfer(_amount);
    }

    function setWtrxAddress(address payable _address) onlyController external {
        wtrxAddress = _address;
    }


    function setFactory(address _factory) onlyController external {
        require(setFactoryManagerCount < 2, "limit factory address");
        factory = _factory;
        setFactoryManagerCount = setFactoryManagerCount.add(1);
    }

    function setDexTokenAmount(uint _amount) onlyController external {
        dexTokenAmount = _amount;
    }

    function swapToken(
        address poolAddress, address tokenIn, address tokenOut
    ) onlyController public returns (uint){
        require(factory != address(0), "factory=0");
        require(IFactory(factory).isBPool(poolAddress), "!pool");
        uint _tokenInAmount = IERC20(tokenIn).balanceOf(address(this));
        IERC20(tokenIn).approve(poolAddress, _tokenInAmount);
        (uint tokenAmountOut,) = IBool(poolAddress).swapExactAmountIn(tokenIn, _tokenInAmount,
tokenOut, 0, MAX);
        return tokenAmountOut;
    }

    function addTokenPoolLiquidity() external {
        require(IERC20(boxToken).balanceOf(msg.sender) >= dexTokenAmount, "not enough
token");
        uint _amount = IERC20(baseToken).balanceOf(address(this));
        uint256 _goverFund = _amount.mul(goverFundDivRate).div(1e18);
        IERC20(baseToken).transfer(governaddr, _goverFund);

        uint _tokenInAmount = _amount.mul(burnRate).div(1e18);

        IERC20(baseToken).approve(basePool, _tokenInAmount);
        IBool(basePool).swapExactAmountIn(baseToken, _tokenInAmount, boxToken, 0, MAX);

        _amount = _amount.sub(_goverFund);
        _amount = _amount.sub(_tokenInAmount);

        IERC20(baseToken).approve(basePool, _amount);
        IBool(basePool).joinswapExternAmountIn(baseToken, _amount, 0);
    }
```

```
    function burnToken() public {
        uint _amount = IERC20(boxToken).balanceOf(address(this));
        if (_amount > 0) {
            RewardToken(boxToken).burn(_amount);
        }
    }

    function setBasePoolToken(address _pool, address _token, address _baseToken) onlyController
public {
        require(factory != address(0), "factory is 0");
        require(IFactory(factory).isBPool(_pool), "!pool");
        require(IBool(_pool).isBound(_token), "not bound");
        require(IBool(_pool).isBound(_baseToken), "not bound");
        require(setBasePoolCount < 2, "limit set base pool");
        setBasePoolCount = setBasePoolCount.add(1);
        basePool = _pool;
        boxToken = _token;
        baseToken = _baseToken;

    }

    function gover(address _goveraddr) onlyController public {
        governaddr = _goveraddr;
    }

    function setBurnRate(uint _rate) onlyController public {
        require(_rate < 5 * 1e17, "< MAX_FEE");
        burnRate = _rate;
    }

    function setGoverFundDivRate(uint256 _goverFundDivRate) onlyController public {
        require(_goverFundDivRate < MAX_FEE, "< MAX_FEE");
        goverFundDivRate = _goverFundDivRate;
    }

    function convert(address _token, address _to) onlyController public {
        address converter = converters[_token];
        Converter(converter).convert(_to);
    }

    function setConverter(
        address _token,
        address _converter
    ) public onlyController {
        converters[_token] = _converter;
    }

}
```

***AssertPriceOracle.sol***

```
pragma solidity ^0.5.9;

import "../lib/Controller.sol";
import "../interface/PriceOracle.sol";
import "../interface/ILinkOracle.sol";
import "../interface/IJustPair.sol";
import "../interface/IPoolView.sol";
import "../CErc20.sol";
import "../lib/SafeMath.sol";

contract AssertPriceOracle is PriceOracle, Controller {
    using SafeMath for uint;

    //underlying address => oracle address
    mapping(address => address) public underlyingOracles;
    //pair address => underlying address
    mapping(address => address) public justLpPairs;
    mapping(address => bool) public justLps;

    // lp => pool view
    mapping(address => address) public abeloLpViews;

    mapping(address => bool) public abeloLps;

    address public wtrxAddress;
    uint public wtrxDecimals = 6;

    constructor(address _controller, address _wtrxAddress) Controller() public {
        setController(_controller);
        wtrxAddress = _wtrxAddress;
    }
```

```
    function toUint(bytes32 inBytes) pure public returns (uint256 outUint) {
        return uint256(inBytes);
    }

    function setUnderlyingOracle(address underlying, address oracle) onlyController external {
        ILinkOracle(oracle).latestAnswer();
        underlyingOracles[underlying] = oracle;
    }


    function getUnderlyingPrice(CToken cToken) external view returns (uint){
        address _underlying = address(CErc20(address(cToken)).underlying());
        if (justLps[_underlying] == true) {
            return calJustLpPrice(_underlying);
        }
        if (abeloLps[_underlying] == true) {
            return calAbeloLpPrice(_underlying);
        }
        address oracle = underlyingOracles[_underlying];
        require(oracle != address(0), "oracle is 0");
        return toUint(ILinkOracle(oracle).latestAnswer());

    }

    function addAbeloPairView(address _lpAddress, address _poolView) onlyController external {
        //        require(IPoolView(_poolView).pool() != _lpAddress, "pool is error");
        abeloLps[_lpAddress] = true;
        abeloLpViews[_lpAddress] = _poolView;


    }

    function addJustLp(address _underlying, address _pairaddress) onlyController external {
        require(_underlying == IJustPair(_pairaddress).tokenAddress(), "pair is error");
        justLps[_pairaddress] = true;
        justLpPairs[_pairaddress] = _underlying;
    }

    function calJustLpPrice(address _pairAddress) public view returns (uint){
        address _underlying = justLpPairs[_pairAddress];

        require(underlyingOracles[_underlying] != address(0), "oracle is 0");

        uint _totalSupply = IJustPair(_pairAddress).totalSupply().div(10 **
IJustPair(_pairAddress).decimals());

        uint _totalUnderlying = ERC20Detailed(_underlying).balanceOf(_pairAddress);
        uint _totalTrx = _pairAddress.balance;
        uint _underlyingDecimal = ERC20Detailed(_underlying).decimals();

        uint lpPerUnderlying = _totalUnderlying.div(_totalSupply).div(10 **
_underlyingDecimal);

        uint _lpPerTrx = _totalTrx.div(_totalSupply).div(10 ** wtrxDecimals);
        address oracle = underlyingOracles[_underlying];

        uint _underLyingPrice = uint(ILinkOracle(oracle).latestAnswer());

        address wtrxOracle = underlyingOracles[wtrxAddress];

        uint _trxPrice = uint(ILinkOracle(wtrxOracle).latestAnswer());

        return _lpPerUnderlying.mul(_underLyingPrice).add(_lpPerTrx.mul(_trxPrice));

    }

    function calAbeloLpPrice(address _lpAddress) public view returns (uint){

        address _poolView = abeloLpViews[_lpAddress];
        uint total;
        address[] memory tokens = IPoolView(_poolView).getCurrentTokens();

        for (uint i = 0; i < 2; i++) {
            address token = tokens[i];
            uint _lpPerToken = IPoolView(_poolView).getTokenAmountPerLp(token);
            address oracle = underlyingOracles[token];
            uint _tokenPrice = uint(ILinkOracle(oracle).latestAnswer());
            total = total.add(_lpPerToken.mul(_tokenPrice));
        }
        return total;
    }


}
```

***SimplePriceOracle.sol***

```
pragma solidity ^0.5.9;

import "../interface/PriceOracle.sol";
import "../CErc20.sol";

contract SimplePriceOracle is PriceOracle {
    mapping(address => uint) prices;

    event PricePosted(address asset, uint previousPriceMantissa, uint requestedPriceMantissa, uint newPriceMantissa);

    function getUnderlyingPrice(CToken cToken) public view returns (uint) {
        if (compareStrings(cToken.symbol(), "cETH")) {
            return 1e18;
        } else {
            return prices[address(CErc20(address(cToken)).underlying())];
        }
    }

    function setUnderlyingPrice(CToken cToken, uint underlyingPriceMantissa) public {
        address asset = address(CErc20(address(cToken)).underlying());
        emit PricePosted(asset, prices[asset], underlyingPriceMantissa, underlyingPriceMantissa);
        prices[asset] = underlyingPriceMantissa;
    }

    function setDirectPrice(address asset, uint price) public {
        emit PricePosted(asset, prices[asset], price, price);
        prices[asset] = price;
    }

    // v1 price oracle interface for use as backing of proxy
    function assetPrices(address asset) external view returns (uint) {
        return prices[asset];
    }

    function compareStrings(string memory a, string memory b) internal pure returns (bool) {
        return (keccak256(abi.encodePacked((a))) == keccak256(abi.encodePacked((b))));
    }
}
```

***IRewardPool.sol***

```
pragma solidity ^0.5.9;

interface IRewardPool {
    function notifyRewardAmount(uint256 reward) external;
}
```

***LPTokenWrapper.sol***

```
pragma solidity ^0.5.0;

import "../lib/SafeMath.sol";
import "../lib/ERC20.sol";

contract LPTokenWrapper {
    using SafeMath for uint256;

    IERC20 public lpToken;

    uint256 private _totalSupply;
    mapping(address => uint256) private _balances;

    function totalSupply() public view returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account) public view returns (uint256) {
        return _balances[account];
    }

    function stake(uint256 amount) public {
        _totalSupply = _totalSupply.add(amount);
        _balances[msg.sender] = _balances[msg.sender].add(amount);
        lpToken.transferFrom(msg.sender, address(this), amount);
    }

    function withdraw(uint256 amount) public {
        _totalSupply = _totalSupply.sub(amount);
        _balances[msg.sender] = _balances[msg.sender].sub(amount);
        lpToken.transfer(msg.sender, amount);
    }

    function migrateStakeFor(address target, uint256 amountNewShare) internal    {
```

```
            _totalSupply = _totalSupply.add(amountNewShare);
            _balances[target] = _balances[target].add(amountNewShare);
        }
}
```

***Reservoir.sol***

```
pragma solidity ^0.5.9;

import "../lib/ERC20.sol";
import "../lib/ERC20Detailed.sol";
import "../lib/Controller.sol";
import "../lib/SafeMath.sol";
import "../lib/IERC20.sol";
import "../lib/EnumerableSet.sol";
import "./RewardToken.sol";
import "./IRewardPool.sol";
import "../Comptroller.sol";

contract Reservoir is Controller {
    using SafeMath for uint256;
    using EnumerableSet for EnumerableSet.AddressSet;

    Comptroller public comptroller;

    uint256 poolLastRewardBlock;

    uint256 fundLastRewardBlock;

    struct PoolInfo {
        uint256 allocPoint; // How many allocation points assigned to this pool. dex to distribute per block.
        address contractAddress;
    }


    // 30 days
    uint public duration = 30 * 28800;

    RewardToken public rewardToken;
    // Dev fund (25%, initially)
    uint public teamFundDivRate = 25e16;
    // gover fund (5%, initially)
    uint public goverFundDivRate = 5e16;
    // insurance fund (5%, initially)
    uint public insuranceFundDivRate = 5e16;
    uint public liquidityRate = 3e16;
    uint public comptrollerRate = 62e16;
    uint256 public MAX_RATE = 1e18;

    uint256 public tokenPerBlock = 5 * 1e18;
    // tokens created per block.
    uint256 public tokenPerBlockForReward;
    uint256 public MIN_TOKEN_REWARD = 5 * 1e16;


    EnumerableSet.AddressSet private teamAddrs;

    address public governaddr;
    address public insuranceaddr;


    PoolInfo[] public poolInfo;

    // Total allocation points. Must be the sum of all allocation points in all pools.
    uint256 public totalAllocPoint = 0;
    uint256 decrement = 0;
    // The block number when PICKLE mining starts.
    uint256 public startBlock;
    uint256 public periodEndBlock;

    // Events
    event Recovered(address token, uint256 amount);

    constructor(
        RewardToken _rewardToken,
        uint256 _startBlock,
        uint256 _duration
    ) Controller() public {
        periodEndBlock = _startBlock.add(_duration);
        require(periodEndBlock > block.number, "end is wrong");
        rewardToken = _rewardToken;
        startBlock = _startBlock;
        fundLastRewardBlock = _startBlock;
```

```
        duration = _duration;
        tokenPerBlockForReward = calTokenPerBlock(tokenPerBlock);

    }


    function calTokenPerBlock(uint256 _blockToken) view internal returns (uint256){
        uint256 _devFund = calRate(_blockToken, teamFundDivRate);
        uint256 _goverFund = calRate(_blockToken, goverFundDivRate);
        uint256 _insuranceFund = calRate(_blockToken, insuranceFundDivRate);
        return _blockToken.sub(_devFund).sub(_goverFund).sub(_insuranceFund);
    }

    function mintToken() onlyOwner public {
        rewardToken.mint(address(this));
    }

    function poolLength() external view returns (uint256) {
        return poolInfo.length;
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(
        address _contractAddress,
        uint256 _allocPoint
    ) public onlyController adjustProduct {
        uint256 lastRewardBlock = block.number > startBlock
        ? block.number
        : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo(
            {
            allocPoint : _allocPoint,
            contractAddress : _contractAddress
            }
            )
        );
    }

    function set(
        uint256 _pid,
        uint256 _allocPoint
    ) public onlyController adjustProduct {
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    modifier adjustProduct()    {
        _;
        if (block.number >= periodEndBlock) {
            if (tokenPerBlock > MIN_TOKEN_REWARD) {
                tokenPerBlock = tokenPerBlock.mul(90).div(100);
            }
            if (tokenPerBlock < MIN_TOKEN_REWARD) {
                tokenPerBlock = MIN_TOKEN_REWARD;
            }

            periodEndBlock = block.number.add(duration);
        }
    }
    function calRate(uint _amount, uint rate) public view returns (uint){
        return _amount.mul(rate).div(1e18);
    }

    // Distribute the output of a cycle in advance
    function distributionToken() adjustProduct public {
        if (block.number < startBlock) {
            return;
        }
        if (poolLastRewardBlock >= periodEndBlock) {
            return;
        }
        uint multiplier = 1;
        if (periodEndBlock > block.number) {
            multiplier = periodEndBlock - block.number;
        }
        uint _reward = multiplier.mul(tokenPerBlock);

        uint goverFund = calRate(_reward, goverFundDivRate);
        uint insuranceFund = calRate(_reward, insuranceFundDivRate);
        uint teamFund = calRate(_reward, teamFundDivRate);
        uint liquidityFund = calRate(_reward, liquidityRate);
```

```
        uint distriReward = _reward.sub(goverFund.add(insuranceFund).add(teamFund).add(liquidityFund));

        safeTokenTransfer(address(comptroller), distriReward);
        Comptroller(comptroller)._setCompRate(calRate(tokenPerBlock, comptrollerRate));

        uint length = poolInfo.length;
        for (uint pid = 0; pid < length; ++pid) {
            PoolInfo storage pool = poolInfo[pid];
            uint256 poolReward = liquidityFund.mul(pool.allocPoint).div(totalAllocPoint);
            safeTokenTransfer(pool.contractAddress, poolReward);
            IRewardPool(pool.contractAddress).notifyRewardAmount(poolReward);
        }
        poolLastRewardBlock = periodEndBlock;


    }

    // Safe pickle transfer function, just in case if rounding error causes pool to not have enough dex.
    function safeTokenTransfer(address _to, uint256 _amount) internal {
        uint256 pickleBal = rewardToken.balanceOf(address(this));
        if (_amount > pickleBal) {
            rewardToken.transfer(_to, pickleBal);
        } else {
            rewardToken.transfer(_to, _amount);
        }
    }

    // Distribute tokens according to teh speed of the block
    function claimFund() onlyController external {
        if (block.number <= fundLastRewardBlock) {
            return;
        }
        uint256 multiplier = block.number - fundLastRewardBlock;
        uint256 boxReward = multiplier.mul(tokenPerBlock);
        uint goverFund = calRate(boxReward, goverFundDivRate);
        uint insuranceFund = calRate(boxReward, insuranceFundDivRate);
        uint teamFund = calRate(boxReward, teamFundDivRate);
        rewardToken.transfer(governaddr, goverFund);
        rewardToken.transfer(insuranceaddr, insuranceFund);
        uint256 perDevFund = teamFund.div(teamAddrs.length());
        for (uint256 i = 0; i < teamAddrs.length() - 1; i++) {
            rewardToken.transfer(teamAddrs.get(i), perDevFund);
        }
        uint256 remainFund = teamFund.sub(perDevFund.mul(teamAddrs.length() - 1));
        rewardToken.transfer(teamAddrs.get(teamAddrs.length() - 1), remainFund);
        goverFund = 0;
        insuranceFund = 0;
        teamFund = 0;
        fundLastRewardBlock = block.number;
        distributionToken();

    }


    function setComptroller(address _address) onlyController external {
        comptroller = Comptroller(_address);
    }

    function addDev(address _devaddr) onlyController public {
        require(teamAddrs.length() < 50, "less 50");
        teamAddrs.add(_devaddr);
    }

    function removeDev(address _devaddr) onlyController public {
        teamAddrs.remove(_devaddr);
    }

    function contains(address _dev) public view returns (bool){
        return teamAddrs.contains(_dev);
    }

    function length() public view returns (uint256){
        return teamAddrs.length();
    }

    function setTeamFundDivRate(uint256 _teamFundDivRate) onlyController public {
        require(_teamFundDivRate <= 25e16, "rate too large");
        teamFundDivRate = _teamFundDivRate;
        comptrollerRate
        =
MAX_RATE.sub(teamFundDivRate).sub(goverFundDivRate).sub(insuranceFundDivRate).sub(liquidityRate);
    }

    function setLiquidityRate(uint _rate) onlyController external {
        require(_rate <= 10e16, "rate too large");
```

```
            liquidityRate = _rate;
            comptrollerRate =
MAX_RATE.sub(teamFundDivRate).sub(goverFundDivRate).sub(insuranceFundDivRate).sub(liquidity
Rate);
        }

        function insurance(address _insuranceaddr) onlyController public {
            insuranceaddr = _insuranceaddr;
        }

        function setInsuranceFundDivRate(uint256 _insuranceFundDivRate) onlyController public {
            require(_insuranceFundDivRate <= 5e16, "rate too large");
            insuranceFundDivRate = _insuranceFundDivRate;
            comptrollerRate
            =
MAX_RATE.sub(teamFundDivRate).sub(goverFundDivRate).sub(insuranceFundDivRate).sub(liquidity
Rate);
        }

        function gover(address _goveraddr) onlyController public {
            governaddr = _goveraddr;
        }

        function setGoverFundDivRate(uint256 _goverFundDivRate) onlyController public {
            require(_goverFundDivRate <= 5e16, "rate too large");
            goverFundDivRate = _goverFundDivRate;
            comptrollerRate
            =
MAX_RATE.sub(teamFundDivRate).sub(goverFundDivRate).sub(insuranceFundDivRate).sub(liquidity
Rate);
        }

}
```

***RewardPool.sol***

```
pragma solidity ^0.5.9;

import "../lib/ERC20.sol";
import "../lib/ERC20Detailed.sol";
import "../lib/Controller.sol";
import "../lib/Math.sol";
import "../lib/SafeMath.sol";
import "../lib/IERC20.sol";
import "../lib/EnumerableSet.sol";
import "./RewardToken.sol";
import "./LPTokenWrapper.sol";
import "../lib/Controller.sol";


contract RewardPool is LPTokenWrapper, Controller {

    IERC20 public rewardToken;
    uint256 public duration; // making it not a constant is less gas efficient, but portable

    uint256 public periodFinish = 0;
    uint256 public rewardRate = 0;
    uint256 public lastUpdateTime;
    uint256 public rewardPerTokenStored;

    uint256 public exitFee = 3e16;
    address public feeManager;

    address public reservoirAddress;

    mapping(address => uint256) public userRewardPerTokenPaid;
    mapping(address => uint256) public rewards;

    mapping(address => bool) smartContractStakers;


    event RewardAdded(uint256 reward);
    event Staked(address indexed user, uint256 amount);
    event Withdrawn(address indexed user, uint256 amount);
    event RewardPaid(address indexed user, uint256 reward);
    event RewardDenied(address indexed user, uint256 reward);
    event SmartContractRecorded(address indexed smartContractAddress, address indexed
smartContractInitiator);

    // Harvest Migration
    event Migrated(address indexed account, uint256 legacyShare, uint256 newShare);

    modifier updateReward(address account) {
        rewardPerTokenStored = rewardPerToken();
        lastUpdateTime = lastTimeRewardApplicable();
        if (account != address(0)) {
```

```
            rewards[account] = earned(account);
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
        _;
    }

    constructor(
        address _reservoirAddress,
        address _rewardToken,
        address _lpToken,
        uint256 _duration,
        address _controller,
        address _feeManager
    ) public
    Controller(){
        reservoirAddress = _reservoirAddress;
        rewardToken = IERC20(_rewardToken);
        lpToken = IERC20(_lpToken);
        feeManager = _feeManager;
        duration = _duration;
        setController(_controller);
    }

    function setExitFee(uint _fee) public onlyController {
        require(_fee <= 1e17, "fee too much");
        exitFee = _fee;
    }

    function lastTimeRewardApplicable() public view returns (uint256) {
        return Math.min(block.number, periodFinish);
    }

    function rewardPerToken() public view returns (uint256) {
        if (totalSupply() == 0) {
            return rewardPerTokenStored;
        }
        return
        rewardPerTokenStored.add(
            lastTimeRewardApplicable()
            .sub(lastUpdateTime)
            .mul(rewardRate)
            .mul(1e18)
            .div(totalSupply())
        );
    }

    function earned(address account) public view returns (uint256) {
        return
        balanceOf(account)
        .mul(rewardPerToken().sub(userRewardPerTokenPaid[account]))
        .div(1e18)
        .add(rewards[account]);
    }

    // stake visibility is public as overriding LPTokenWrapper's stake() function
    function stake(uint256 amount) public updateReward(msg.sender) {
        require(amount > 0, "Cannot stake 0");
        super.stake(amount);
        emit Staked(msg.sender, amount);
    }

    function withdraw(uint256 amount) public updateReward(msg.sender) {
        require(amount > 0, "Cannot withdraw 0");
        uint exitAmount = amount.mul(exitFee).div(1e18);
        uint _amount = amount.sub(exitAmount);
        super.withdraw(_amount);
        emit Withdrawn(msg.sender, amount);
    }

    function exit() external {
        withdraw(balanceOf(msg.sender));
        getReward();
    }

    function pushReward(address recipient) public updateReward(recipient) onlyController {
        uint256 reward = earned(recipient);
        if (reward > 0) {
            rewards[recipient] = 0;
            rewardToken.transfer(recipient, reward);
            emit RewardPaid(recipient, reward);

        }
    }

    function getReward() public updateReward(msg.sender) {
        uint256 reward = earned(msg.sender);
        if (reward > 0) {
```

```
                rewards[msg.sender] = 0;
                // If it is a normal user and not smart contract,
                // then the requirement will pass
                // If it is a smart contract, then
                // make sure that it is not on our greyList.
                if (tx.origin == msg.sender) {
                    rewardToken.transfer(msg.sender, reward);
                    emit RewardPaid(msg.sender, reward);
                } else {
                    emit RewardDenied(msg.sender, reward);
                }
            }
        }

        function notifyRewardAmount(uint256 reward)
        external
        updateReward(address(0))
        {
            require(msg.sender == controller() || msg.sender == reservoirAddress, "only controller and reservoir");

            require(reward < uint(- 1) / 1e18, "the notified reward cannot invoke multiplication overflow");

            if (block.number >= periodFinish) {
                rewardRate = reward.div(duration);
            } else {
                uint256 remaining = periodFinish.sub(block.number);
                uint256 leftover = remaining.mul(rewardRate);
                rewardRate = reward.add(leftover).div(duration);
            }
            lastUpdateTime = block.number;
            periodFinish = block.number.add(duration);
            emit RewardAdded(reward);
        }

        function setReservoir(address _reservoir) onlyController public {
            require(_reservoir != address(0), "reservoir is not 0");
            reservoirAddress = _reservoir;

        }
}
```

***RewardToken,.sol***

```
pragma solidity ^0.5.9;
pragma experimental ABIEncoderV2;

import "../lib/ERC20.sol";
import "../lib/ERC20Detailed.sol";
import "../lib/Ownable.sol";
import "../lib/ERC20Burnable.sol";

contract RewardToken is ERC20, ERC20Detailed, ERC20Burnable, Ownable {

    uint public constant _totalSupply = 43200000e18;


    constructor () public ERC20Detailed("BOX-BANK", "BOX", 18) {
    }

    function mint(address _to) public onlyOwner {
        require(totalSupply() == 0, "mint once");
        _mint(_to, _totalSupply);
    }

}
```

***BaseJumpRateModelV2.sol***

```
pragma solidity ^0.5.9;

import "./lib/SafeMath.sol";

/**
  * @title Logic for Compound's JumpRateModel Contract V2.
  * @author Compound (modified by Dharma Labs, refactored by Arr00)
  * @notice Version 2 modifies Version 1 by enabling updateable parameters.
  */
contract BaseJumpRateModelV2 {
    using SafeMath for uint;

    event NewInterestParams(uint baseRatePerBlock, uint multiplierPerBlock, uint jumpMultiplierPerBlock, uint kink);
```

```
    /**
     * @notice The address of the owner, i.e. the Timelock contract, which can update parameters directly
     */
    address public owner;

    /**
     * @notice The approximate number of blocks per year that is assumed by the interest rate model
     */
    uint public constant blocksPerYear = 10512000;

    /**
     * @notice The multiplier of utilization rate that gives the slope of the interest rate
     */
    uint public multiplierPerBlock;

    /**
     * @notice The base interest rate which is the y-intercept when utilization rate is 0
     */
    uint public baseRatePerBlock;

    /**
     * @notice The multiplierPerBlock after hitting a specified utilization point
     */
    uint public jumpMultiplierPerBlock;

    /**
     * @notice The utilization point at which the jump multiplier is applied
     */
    uint public kink;

    /**
     * @notice Construct an interest rate model
     * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
     * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
     * @param jumpMultiplierPerYear The multiplierPerBlock after hitting a specified utilization point
     * @param kink_ The utilization point at which the jump multiplier is applied
     * @param owner_ The address of the owner,
     *               i.e. the Timelock contract (which has the ability to update parameters directly)
     */
    constructor(
        uint baseRatePerYear,
        uint multiplierPerYear,
        uint jumpMultiplierPerYear,
        uint kink_, address owner_
    ) internal {
        owner = owner_;

        updateJumpRateModelInternal(baseRatePerYear, multiplierPerYear, jumpMultiplierPerYear, kink_);
    }

    /**
     * @notice Update the parameters of the interest rate model (only callable by owner, i.e. Timelock)
     * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
     * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
     * @param jumpMultiplierPerYear The multiplierPerBlock after hitting a specified utilization point
     * @param kink_ The utilization point at which the jump multiplier is applied
     */
    function updateJumpRateModel(
        uint baseRatePerYear,
        uint multiplierPerYear,
        uint jumpMultiplierPerYear,
        uint kink_
    ) external {
        require(msg.sender == owner, "only the owner may call this function.");

        updateJumpRateModelInternal(baseRatePerYear, multiplierPerYear, jumpMultiplierPerYear, kink_);
    }

    /**
     * @notice Calculates the utilization rate of the market: `borrows / (cash + borrows - reserves)`
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market (currently unused)
     * @return The utilization rate as a mantissa between [0, 1e18]
     */
    function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }
```

```
            return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
        }

        /**
          * @notice Calculates the current borrow rate per block, with the error code expected by the market
          * @param cash The amount of cash in the market
          * @param borrows The amount of borrows in the market
          * @param reserves The amount of reserves in the market
          * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
          */
        function getBorrowRateInternal(uint cash, uint borrows, uint reserves) internal view returns (uint) {
            uint util = utilizationRate(cash, borrows, reserves);

            if (util <= kink) {
                return util.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
            } else {
                uint normalRate = kink.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
                uint excessUtil = util.sub(kink);
                return excessUtil.mul(jumpMultiplierPerBlock).div(1e18).add(normalRate);
            }
        }

        /**
          * @notice Calculates the current supply rate per block
          * @param cash The amount of cash in the market
          * @param borrows The amount of borrows in the market
          * @param reserves The amount of reserves in the market
          * @param reserveFactorMantissa The current reserve factor for the market
          * @return The supply rate percentage per block as a mantissa (scaled by 1e18)
          */
        function getSupplyRate(
            uint cash,
            uint borrows,
            uint reserves,
            uint reserveFactorMantissa
        ) public view returns (uint) {
            uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
            uint borrowRate = getBorrowRateInternal(cash, borrows, reserves);
            uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
            return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
        }

        /**
          * @notice Internal function to update the parameters of the interest rate model
          * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
          * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
          * @param jumpMultiplierPerYear The multiplierPerBlock after hitting a specified utilization point
          * @param kink_ The utilization point at which the jump multiplier is applied
          */
        function updateJumpRateModelInternal(
            uint baseRatePerYear,
            uint multiplierPerYear,
            uint jumpMultiplierPerYear,
            uint kink_
        ) internal {
            baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
            multiplierPerBlock = (multiplierPerYear.mul(1e18)).div(blocksPerYear.mul(kink_));
            jumpMultiplierPerBlock = jumpMultiplierPerYear.div(blocksPerYear);
            kink = kink_;

            emit NewInterestParams(baseRatePerBlock, multiplierPerBlock, jumpMultiplierPerBlock, kink);
        }
}
```

***CErc20.sol***

```
pragma solidity ^0.5.9;

import "./CToken.sol";
import "./interface/CErc20Interface.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_, symbol_, decimals_);

                        // Set underlying and sanity check it
                        underlying = underlying_;
                        IERC20(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);
                        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
    ) external returns (uint) {
        (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) {
        IERC20 token = IERC20(underlying);
        return token.balanceOf(address(this));
    }

    /**
     * @dev Similar to EIP20 transfer, except it handles a False result from `transferFrom` and reverts in that case.
     *      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-affected-d67bf08521ca
     */
    function doTransferIn(address from, uint amount) internal returns (uint) {
        EIP20NonStandardInterface token = EIP20NonStandardInterface(underlying);
        uint balanceBefore = IERC20(underlying).balanceOf(address(this));
        token.transferFrom(from, address(this), amount);
        //todo EIP20Non
        //        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_OVERFLOWSFER_IN_FAILED");

        // Calculate the amount that was *actually* transferred
        uint balanceAfter = IERC20(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 an explanatory
     *      error code rather than reverting.
     *      If caller has not called checked protocol's balance, this may revert due to
     *      `insufficient cash held in this contract. If caller has checked protocol's balance prior to this call,
     *      and verified
     *      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-affected-d67bf08521ca
        */
    //todo EIP20Non for tron usdt
    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");
    }
}
```

***CErc20Delegate.sol***

```
pragma solidity ^0.5.9;

import "./CErc20.sol";
import "./interface/CDelegateInterface.sol";

/**
 * @title Compound's CErc20Delegate Contract
 * @notice CTokens which wrap an EIP-20 underlying and are delegated to
 * @author Compound
 */
contract CErc20Delegate is CErc20, CDelegateInterface {
    /**
     * @notice Construct an empty delegate
     */
    constructor() public {}

    /**
     * @notice Called by the delegator on a delegate to initialize it for duty
     * @param data The encoded bytes data for any initialization
     */
    function _becomeImplementation(bytes memory data) public {
        // Shh -- currently unused
        data;

        // Shh -- we don't ever want this hook to be marked pure
        if (false) {
            implementation = address(0);
        }

        require(msg.sender == admin, "only the admin may call _becomeImplementation");
    }

    /**
     * @notice Called by the delegator on a delegate to forfeit its responsibility
     */
    function _resignImplementation() public {
        // Shh -- we don't ever want this hook to be marked pure
        if (false) {
            implementation = address(0);
        }

        require(msg.sender == admin, "only the admin may call _resignImplementation");
    }
}
```

***CErc20Delegator.sol***

```
pragma solidity ^0.5.9;

import "./interface/CTokenInterface.sol";
import "./interface/CDelegatorInterface.sol";
import "./interface/CErc20Interface.sol";

/**
 * @title Compound's CErc20Delegator Contract
 * @notice CTokens which wrap an EIP-20 underlying and delegate to an implementation
```

```
 * @author Compound
 */
contract CErc20Delegator is CTokenInterface, CErc20Interface, CDelegatorInterface {
    /**
     * @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
     * @param implementation_ The address of the implementation the contract delegates to
     * @param becomeImplementationData The encoded args for becomeImplementation
     */
    constructor(address underlying_,
                ComptrollerInterface comptroller_,
                InterestRateModel interestRateModel_,
                uint initialExchangeRateMantissa_,
                string memory name_,
                string memory symbol_,
                uint8 decimals_,
                address payable admin_,
                address implementation_,
                bytes memory becomeImplementationData) public {
        // Creator of the contract is admin during initialization
        admin = msg.sender;

        // First delegate gets to initialize the delegator (i.e. storage contract)
        delegateTo(implementation_,
abi.encodeWithSignature("initialize(address,address,address,uint256,string,string,uint8)",
                           underlying_,
                           comptroller_,
                           interestRateModel_,
                           initialExchangeRateMantissa_,
                           name_,
                           symbol_,
                           decimals_));

        // New implementations always get set via the settor (post-initialize)
        _setImplementation(implementation_, false, becomeImplementationData);

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

    /**
     * @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 implementation
     * @param becomeImplementationData The encoded bytes data to be passed to _becomeImplementation
     */
    function _setImplementation(
        address implementation_,
        bool allowResign,
        bytes memory becomeImplementationData
    ) public {
        require(msg.sender == admin, "CErc20Delegator::_setImplementation: Caller must be admin");

        if (allowResign) {
            delegateToImplementation(abi.encodeWithSignature("_resignImplementation()"));
        }

        address oldImplementation = implementation;
        implementation = implementation_;

        delegateToImplementation(abi.encodeWithSignature("_becomeImplementation(bytes)",
becomeImplementationData));

        emit NewImplementation(oldImplementation, implementation);
    }

    /**
     * @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) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("mint(uint256)",
mintAmount));
```

```
        return abi.decode(data, (uint));
    }

    /**
      * @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) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("redeem(uint256)", redeemTokens));
        return abi.decode(data, (uint));
    }

    /**
      * @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) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("redeemUnderlying(uint256)", redeemAmount));
        return abi.decode(data, (uint));
    }

    /**
      * @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) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("borrow(uint256)", borrowAmount));
        return abi.decode(data, (uint));
    }

    /**
      * @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) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("repayBorrow(uint256)", repayAmount));
        return abi.decode(data, (uint));
    }

    /**
      * @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) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("repayBorrowBehalf(address,uint256)", borrower, repayAmount));
        return abi.decode(data, (uint));
    }

    /**
      * @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 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function liquidateBorrow(
        address borrower,
        uint repayAmount,
        CTokenInterface cTokenCollateral
    ) external returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature(
                "liquidateBorrow(address,uint256,address)",
                borrower,
                repayAmount,
                cTokenCollateral
            )
        );
        return abi.decode(data, (uint));
    }

    /**
```

```
     * @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, uint amount) external returns (bool) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("transfer(address,uint256)", dst, amount));
        return abi.decode(data, (bool));
    }

    /**
     * @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) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("transferFrom(address,address,uint256)", src, dst, amount)
        );
        return abi.decode(data, (bool));
    }

    /**
     * @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) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("approve(address,uint256)", spender, amount)
        );
        return abi.decode(data, (bool));
    }

    /**
     * @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 (uint) {
        bytes memory data = delegateToViewImplementation(
            abi.encodeWithSignature("allowance(address,address)", owner, spender)
        );
        return abi.decode(data, (uint));
    }

    /**
     * @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 (uint) {
        bytes memory data = delegateToViewImplementation(abi.encodeWithSignature("balanceOf(address)", owner));
        return abi.decode(data, (uint));
    }

    /**
     * @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) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("balanceOfUnderlying(address)", owner));
        return abi.decode(data, (uint));
    }


    function balanceOfUnderlyingView(address owner) external view returns (uint) {
        bytes memory data = delegateToViewImplementation(
            abi.encodeWithSignature("balanceOfUnderlyingView(address)", owner)
        );
        return abi.decode(data, (uint));
    }

    /**
     * @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) {
        bytes memory data = delegateToViewImplementation(
            abi.encodeWithSignature("getAccountSnapshot(address)", account)
        );
        return abi.decode(data, (uint, uint, uint, uint));
    }

    /**
     * @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) {
        bytes memory data = delegateToViewImplementation(abi.encodeWithSignature("borrowRatePerBlock()"));
        return abi.decode(data, (uint));
    }

    /**
     * @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) {
        bytes memory data = delegateToViewImplementation(abi.encodeWithSignature("supplyRatePerBlock()"));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Returns the current total borrows plus accrued interest
     * @return The total borrows with interest
     */
    function totalBorrowsCurrent() external returns (uint) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("totalBorrowsCurrent()"));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Accrue interest to updated borrowIndex
     *           and then calculate account's borrow balance using the updated borrowIndex
     * @param account The address whose balance should be calculated after updating borrowIndex
     * @return The calculated balance
     */
    function borrowBalanceCurrent(address account) external returns (uint) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("borrowBalanceCurrent(address)", account));
        return abi.decode(data, (uint));
    }

    /**
     * @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) {
        bytes memory data = delegateToViewImplementation(
            abi.encodeWithSignature("borrowBalanceStored(address)", account)
        );
        return abi.decode(data, (uint));
    }

    /**
      * @notice Accrue interest then return the up-to-date exchange rate
      * @return Calculated exchange rate scaled by 1e18
      */
    function exchangeRateCurrent() public returns (uint) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("exchangeRateCurrent()"));
        return abi.decode(data, (uint));
    }

    /**
     * @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) {
        bytes memory data = delegateToViewImplementation(abi.encodeWithSignature("exchangeRateStored()"));
        return abi.decode(data, (uint));
    }

    /**
     * @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) {
        bytes memory data = delegateToViewImplementation(abi.encodeWithSignature("getCash()"));
        return abi.decode(data, (uint));
    }

    /**
     * @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) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("accrueInterest()"));
        return abi.decode(data, (uint));
    }

    /**
     * @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 returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("seize(address,address,uint256)", liquidator, borrower, seizeTokens)
        );
        return abi.decode(data, (uint));
    }

    /*** Admin Functions ***/

    /**
     * @notice Begins transfer of admin rights.
     *      The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
     * @dev Admin function to begin change of admin.
     *   The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
     * @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) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("_setPendingAdmin(address)", newPendingAdmin)
        );
        return abi.decode(data, (uint));
    }

    /**
     * @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) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("_setComptroller(address)",
            newComptroller));
        return abi.decode(data, (uint));
    }

    function _setFeeManager(address payable _feeManager) external returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("_setFeeManager(address)",
            _feeManager));
        return abi.decode(data, (uint));
    }

    function flashLoan(
        address _receiver,
        address _reserve,
        uint256 _amount,
        bytes memory _params
    ) public returns (uint){
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("flashLoan(address,address,uint256,bytes)",
              _receiver, _reserve, _amount, _params));
        return abi.decode(data, (uint));
    }

    /**
     * @notice accrues interest and sets a new reserve factor for the protocol using _setReserveFactorFresh
```

```
     * @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 returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("_setReserveFactor(uint256)", newReserveFactorMantissa)
        );
        return abi.decode(data, (uint));
    }

    /**
     * @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) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("_acceptAdmin()"));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Accrues interest and adds reserves by transferring from admin
     * @param addAmount Amount of reserves to add
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _addReserves(uint addAmount) external returns (uint) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("_addReserves(uint256)", addAmount));
        return abi.decode(data, (uint));
    }

    /**
     * @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 returns (uint) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("_reduceReserves(uint256)", reduceAmount));
        return abi.decode(data, (uint));
    }

    /**
     * @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) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("_setInterestRateModel(address)",
            newInterestRateModel));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Internal method to delegate execution to another contract
     * @dev It returns to the external caller whatever the implementation returns or forwards reverts
     * @param callee The contract to delegatecall
     * @param data The raw data to delegatecall
     * @return The returned bytes from the delegatecall
     */
    function delegateTo(address callee, bytes memory data) internal returns (bytes memory) {
        (bool success, bytes memory returnData) = callee.delegatecall(data);
        assembly {
            if eq(success, 0) {
                revert(add(returnData, 0x20), returndatasize)
            }
        }
        return returnData;
    }

    /**
     * @notice Delegates execution to the implementation contract
     * @dev It returns to the external caller whatever the implementation returns or forwards reverts
     * @param data The raw data to delegatecall
     * @return The returned bytes from the delegatecall
     */
    function delegateToImplementation(bytes memory data) public returns (bytes memory) {
        return delegateTo(implementation, data);
    }

    /**
     * @notice Delegates execution to an implementation contract
     * @dev It returns to the external caller whatever the implementation returns or forwards reverts
```

```
     *  There are an additional 2 prefix uints from the wrapper returndata, which we ignore since we make an extra hop.
     * @param data The raw data to delegatecall
     * @return The returned bytes from the delegatecall
     */
    function delegateToViewImplementation(bytes memory data) public view returns (bytes memory) {
        (bool success, bytes memory returnData) = address(this).staticcall(
            abi.encodeWithSignature("delegateToImplementation(bytes)", data)
        );
        assembly {
            if eq(success, 0) {
                revert(add(returnData, 0x20), returndatasize)
            }
        }
        return abi.decode(returnData, (bytes));
    }

    /**
     * @notice Delegates execution to an implementation contract
     * @dev It returns to the external caller whatever the implementation returns or forwards reverts
     */
    function() external payable {
        require(msg.value == 0, "CErc20Delegator:fallback: cannot send value to fallback");

        // delegate all other functions to current implementation
        (bool success,) = implementation.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)}
        }
    }
}
```

***CErc20Immutable.sol***

```
pragma solidity ^0.5.9;

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_,
            symbol_,
            decimals_
        );

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

***CNative.sol***

```
pragma solidity ^0.5.9;

import "./CToken.sol";
import "./lib/NativeAddressLib.sol";

/**
  * @title Compound's CNative Contract
  * @notice CToken which wraps Ether
  * @author Compound
  */
contract CNative is CToken {
    /**
      * @notice Construct a new CNative 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_ 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(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(comptroller_, interestRateModel_, initialExchangeRateMantissa_, name_, symbol_,
decimals_);

                // Set the proper admin now that initialization is done
                admin = admin_;
                underlying = NativeAddressLib.nativeAddress();

    }


    /*** User Interface ***/

    /**
      * @notice Sender supplies assets into the market and receives cTokens in exchange
      * @dev Reverts upon any failure
      */
    function mint() external payable {
        (uint err,) = mintInternal(msg.value);
        requireNoError(err, "mint failed");
    }

    /**
      * @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
      * @dev Reverts upon any failure
```

```
     */
    function repayBorrow() external payable {
        (uint err,) = repayBorrowInternal(msg.value);
        requireNoError(err, "repayBorrow failed");
    }

    /**
     * @notice Sender repays a borrow belonging to borrower
     * @dev Reverts upon any failure
     * @param borrower the account with the debt being payed off
     */
    function repayBorrowBehalf(address borrower) external payable {
        (uint err,) = repayBorrowBehalfInternal(borrower, msg.value);
        requireNoError(err, "repayBorrowBehalf failed");
    }

    /**
     * @notice The sender liquidates the borrowers collateral.
     *  The collateral seized is transferred to the liquidator.
     * @dev Reverts upon any failure
     * @param borrower The borrower of this cToken to be liquidated
     * @param cTokenCollateral The market in which to seize collateral from the borrower
     */
    function liquidateBorrow(address borrower, CToken cTokenCollateral) external payable {
        (uint err,) = liquidateBorrowInternal(borrower, msg.value, cTokenCollateral);
        requireNoError(err, "liquidateBorrow failed");
    }

    /**
     * @notice Send Ether to CNative to mint
     */
    function() external payable {
        (uint err,) = mintInternal(msg.value);
        requireNoError(err, "mint failed");
    }

    /*** Safe Token ***/

    /**
     * @notice Gets balance of this contract in terms of Ether, before this message
     * @dev This excludes the value of the current message, if any
     * @return The quantity of Ether owned by this contract
     */
    function getCashPrior() internal view returns (uint) {
        (MathError err, uint startingBalance) = subUInt(address(this).balance, msg.value);
        require(err == MathError.NO_ERROR);
        return startingBalance;
    }

    /**
     * @notice Perform the actual transfer in, which is a no-op
     * @param from Address sending the Ether
     * @param amount Amount of Ether being sent
     * @return The actual amount of Ether transferred
     */
    function doTransferIn(address from, uint amount) internal returns (uint) {
        // Sanity checks
        require(msg.sender == from, "sender mismatch");
        require(msg.value == amount, "value mismatch");
        return amount;
    }

    function doTransferOut(address payable to, uint amount) internal {
        /* Send the Ether, with minimal gas and revert on failure */
        to.transfer(amount);
    }

    function requireNoError(uint errCode, string memory message) internal pure {
        if (errCode == uint(Error.NO_ERROR)) {
            return;
        }

        bytes memory fullMessage = new bytes(bytes(message).length + 5);
        uint i;

        for (i = 0; i < bytes(message).length; i++) {
            fullMessage[i] = bytes(message)[i];
        }

        fullMessage[i + 0] = byte(uint8(32));
        fullMessage[i + 1] = byte(uint8(40));
        fullMessage[i + 2] = byte(uint8(48 + (errCode / 10)));
        fullMessage[i + 3] = byte(uint8(48 + (errCode % 10)));
        fullMessage[i + 4] = byte(uint8(41));

        require(errCode == uint(Error.NO_ERROR), string(fullMessage));
    }
```

```
}
```

**Comptroller.sol**

```
pragma solidity ^0.5.9;

import "./CToken.sol";
import "./ErrorReporter.sol";
import "./lib/Exponential.sol";
import "./interface/PriceOracle.sol";
import "./interface/ComptrollerInterface.sol";
import "./interface/ComptrollerStorage.sol";
import "./Unitroller.sol";
import "./token/RewardToken.sol";


/**
  * @title Compound's Comptroller Contract
  * @author Compound
  */
contract Comptroller is ComptrollerV4Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponential {
    /// @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 newCollateralFactorMantissa);

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

    /// @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 supplySpeed, uint borrowSpeed);

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

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

    /// @notice Emitted when borrow cap for a cToken is changed
    event NewBorrowCap(CToken indexed cToken, uint newBorrowCap);

    /// @notice Emitted when borrow cap guardian is changed
    event NewBorrowCapGuardian(address oldBorrowCapGuardian, address newBorrowCapGuardian);

    /// @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

uint public constant DEFAULT_TOKEN_WEIGHT = 100;
uint public constant MAX_TOKEN_WEIGHT = 500;

address private riskControl;

constructor() public {
        admin = msg.sender;
        rewardAddress = 0xc00e94Cb662C3520282E6f5717214004A7f26888;
}

/*** 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 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
        minter;
        mintAmount;

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

        // 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) external {
        // 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 (uint) {
        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 returns (uint) {
        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(cToken),
            redeemTokens,
            0);
        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) external {
            // 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 market
         * @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 (uint) {
            // 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]);
            }

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


            uint borrowCap = borrowCaps[cToken];
            // Borrow cap of 0 corresponds to unlimited borrowing
            if (borrowCap != 0) {
                uint totalBorrows = CToken(cToken).totalBorrows();
                (MathError mathErr, uint nextTotalBorrows) = addUInt(totalBorrows, borrowAmount);
                require(mathErr == MathError.NO_ERROR, "total borrows overflow");
                require(nextTotalBorrows < borrowCap, "market borrow cap reached");
            }

            (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(
                borrower,
                CToken(cToken),
                0,
                borrowAmount);
            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);
        }

        // 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);
        }

        /* 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}), borrowBalance);
        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);
        }

        // 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.sol)
     */
    function transferAllowed(address cToken, address src, address dst, uint transferTokens) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!transferGuardianPaused, "transfer is paused");

        // 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, CToken(0), 0, 0);

        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/borrowed
     * @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,
            CToken(cTokenModify),
            redeemTokens,
            borrowAmount);
        return (uint(err), liquidity, shortfall);
    }

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borrowed
     * @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 data,
     *  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.getAccountSnapshot(account);
            if (oErr != 0) {// semi-opaque error code, we assume NO_ERROR == 0 is invariant between upgrades
                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.oraclePrice);
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }

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

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

                // borrow effect
                // sumBorrowPlusEffects += oraclePrice * borrowAmount
                (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(
                    vars.oraclePrice,
                    borrowAmount,
                    vars.sumBorrowPlusEffects);
                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 cTokenCollateral tokens
      * @return (errorCode, number of cTokenCollateral tokens to be seized in a liquidation)
      */
    function liquidateCalculateSeizeTokens(
        address cTokenBorrowed,
        address cTokenCollateral,
        uint actualRepayAmount
    ) external view returns (uint, uint) {
        /* 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 * exchangeRate)
          */
        uint exchangeRateMantissa = CToken(cTokenCollateral).exchangeRateStored();

        // Note: reverts on error
        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);
    }

    function getRiskControl() external returns (address){
        return riskControl;
    }

    function _setRiskControl(address _address) external {
        require(msg.sender == admin, "!admin");
        riskControl = _address;
    }

    /**
      * @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 (uint) {
        // 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_VALIDATION);
        }

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

        // 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 : 0,
            weight : DEFAULT_TOKEN_WEIGHT
            });

            _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 Set the given borrow caps for the given cToken markets.
          *       Borrowing that brings total borrows to or above borrow cap will revert.
          * @dev Admin or borrowCapGuardian function to set the borrow caps.
          *   A borrow cap of 0 corresponds to unlimited borrowing.
          * @param cTokens The addresses of the markets (tokens) to change the borrow caps for
          * @param newBorrowCaps The new borrow cap values in underlying to be set.
          *   A value of 0 corresponds to unlimited borrowing.
          */
        function _setMarketBorrowCaps(CToken[] calldata cTokens, uint[] calldata newBorrowCaps)
external {
            require(msg.sender == admin || msg.sender == borrowCapGuardian,
                "only admin or borrow cap guardian can set borrow caps");

            uint numMarkets = cTokens.length;
            uint numBorrowCaps = newBorrowCaps.length;

            require(numMarkets != 0 && numMarkets == numBorrowCaps, "invalid input");

            for (uint i = 0; i < numMarkets; i++) {
                borrowCaps[address(cTokens[i])] = newBorrowCaps[i];
                emit NewBorrowCap(cTokens[i], newBorrowCaps[i]);
            }
        }

        /**
          * @notice Admin function to change the Borrow Cap Guardian
          * @param newBorrowCapGuardian The address of the new Borrow Cap Guardian
          */
        function _setBorrowCapGuardian(address newBorrowCapGuardian) external {
            require(msg.sender == admin, "only admin can set borrow cap guardian");

            // Save current value for inclusion in log
            address oldBorrowCapGuardian = borrowCapGuardian;

            // Store borrowCapGuardian with value newBorrowCapGuardian
            borrowCapGuardian = newBorrowCapGuardian;

            // Emit NewBorrowCapGuardian(OldBorrowCapGuardian, NewBorrowCapGuardian)
            emit NewBorrowCapGuardian(oldBorrowCapGuardian, newBorrowCapGuardian);
        }

        /**
          * @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 can pause");
        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 can pause");
        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 can pause");
        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 can pause");
        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];
            Market memory market = markets[address(cToken)];
            if (market.isComped) {
                Exp memory assetPrice = Exp({mantissa : oracle.getUnderlyingPrice(cToken)});
                Exp memory utility = mul_(assetPrice, cToken.totalBorrows());
                utilities[i] = div_(mul_(utility, market.weight), DEFAULT_TOKEN_WEIGHT);
                totalUtility = add_(totalUtility, utility);
            }
        }
```

```
            for (uint i = 0; i < allMarkets_.length; i++) {
                CToken cToken = allMarkets[i];
                uint comSupplyRate = div_(mul_(compRate, comSupplyRatio), expScale);
                uint comBorrowRate = div_(mul_(compRate, comBorrowRatio), expScale);
                uint newSupplySpeed = totalUtility.mantissa > 0 ? mul_(comSupplyRate, div_(utilities[i], totalUtility)) : 0;
                uint newBorrowSpeed = totalUtility.mantissa > 0 ? mul_(comBorrowRate, div_(utilities[i], totalUtility)) : 0;

                compSpeeds[address(cToken)] = newSupplySpeed;
                compBorrowSpeeds[address(cToken)] = newBorrowSpeed;

                emit CompSpeedUpdated(cToken, newSupplySpeed, newBorrowSpeed);
            }
        }

        /**
          * @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({mantissa : 0});
                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 = compBorrowSpeeds[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({mantissa : 0});
                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 : compClaimThreshold);
```

```
            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,
            bool distributeAll
        ) internal {
            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), marketBorrowIndex);
                uint borrowerDelta = mul_(borrowerAmount, deltaIndex);
                uint borrowerAccrued = add_(compAccrued[borrower], borrowerDelta);
                compAccrued[borrower] = transferComp(borrower, borrowerAccrued, distributeAll ? 0 : compClaimThreshold);
                emit DistributedBorrowerComp(CToken(cToken), borrower, borrowerDelta, borrowIndex.mantissa);
            }
        }

        /**
         * @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) {
                RewardToken comp = RewardToken(rewardAddress);
                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 suppliers) public {
            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() || msg.sender == reservoir, "only admin and reservoir can change comp rate");

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

            refreshCompSpeedsInternal();
        }

        function _setReservoir(address _address) public {
            require(adminOrInitializing(), "only admin can change reservoir");
            reservoir = _address;
        }

        function _setRewardAddress(address _address) public {
            require(adminOrInitializing(), "only admin can change comp address");
            rewardAddress = _address;

        }

        function _setComBorrowRatio(uint compRate_) public {
            require(adminOrInitializing(), "only admin can change comp rate");
            comBorrowRatio = compRate_;
            refreshCompSpeedsInternal();
        }

        function _setComSupplyRatio(uint compRate_) public {
            require(adminOrInitializing(), "only admin can change comp rate");
            comSupplyRatio = 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 _setCompWeight(address cToken, uint _weight) public {
            require(adminOrInitializing(), "only admin can set weight");
            Market storage market = markets[cToken];
            require(market.isListed == true, "comp market is not listed");
            require(market.isComped == true, "comp market is not added");
            require(_weight < MAX_TOKEN_WEIGHT, "weight < max");
            market.weight = _weight;

        }

        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;
            market.weight = DEFAULT_TOKEN_WEIGHT;
            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;
        }

}
```

***CToken.sol***

```
pragma solidity ^0.5.9;

import "./interface/ComptrollerInterface.sol";
import "./interface/CTokenInterface.sol";
import "./ErrorReporter.sol";
import "./lib/Exponential.sol";
import "./lib/IERC20.sol";
import "./interface/EIP20NonStandardInterface.sol";
import "./interface/InterestRateModel.sol";
import "./Comptroller.sol";
import "./RiskController.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 set)
        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/refund)
        _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 (uint) {
        /* 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, allowed);
        }

        /* 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 (bool) {
        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;
    }

    function balanceOfUnderlyingView(address owner) external view returns (uint){
        Exp memory exchangeRate = Exp({mantissa : exchangeRateStored()});
        (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,
reserveFactorMantissa);
    }

    /**
      * @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 using the updated borrowIndex
      * @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 case.
         */
        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, totalReserves);
            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,
reservesPrior);
            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_CALCULATION_FAILED,
                    uint(mathErr)
                );
            }

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

FailureInfo.ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED,
                    uint(mathErr)
                );
            }

            (mathErr, totalBorrowsNew) = addUInt(interestAccumulated, borrowsPrior);
            if (mathErr != MathError.NO_ERROR) {
                return failOpaque(
```

```
                    Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED,
                    uint(mathErr)
                );
            }

            (mathErr, totalReservesNew) = mulScalarTruncateAddUInt(
                Exp({mantissa : reserveFactorMantissa}),
                interestAccumulated,
                reservesPrior
            );
            if (mathErr != MathError.NO_ERROR) {
                return failOpaque(
                    Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED,
                    uint(mathErr)
                );
            }

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

            ////////////////////////
            // 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),
          *                      and the actual mint amount.
          */
        function mintInternal(uint mintAmount) internal nonReentrant returns (uint, uint) {
            address riskControl = Comptroller(address(comptroller)).getRiskControl();
            if (riskControl != address(0)) {
                uint risk = RiskController(riskControl).checkMintRisk(address(this), msg.sender);
                require(risk == 0, 'risk control');
            }
            uint error = accrueInterest();
            if (error != uint(Error.NO_ERROR)) {
                // accrueInterest emits logs on errors, but we still want to log the fact that an attempted
borrow failed
                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 to
            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),
```

```
     *          and the actual mint amount.
     */
    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, allowed), 0);
        }

        /* 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(vars.mathErr)), 0);
        }

        /////////////////////////
        // 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 : vars.exchangeRateMantissa})
        );
        require(vars.mathErr == MathError.NO_ERROR, "MINT_EXCHANGE_CALCULATION_FAILED");

        /*
          * We calculate the new total supply of cTokens and minter token balance, checking for overflow:
          *  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 attempted redeem failed
            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 attempted redeem failed
            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 redeemTokensIn or redeemAmountIn may be non-zero)
     * @param redeemAmountIn The number of underlying tokens to receive from redeeming cTokens
     *          (only one of redeemTokensIn or redeemAmountIn may be non-zero)
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function redeemFresh(address payable redeemer, uint redeemTokensIn, uint redeemAmountIn) internal returns (uint) {
        require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of redeemTokensIn or redeemAmountIn must be zero");

        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(vars.mathErr));
        }

        /* 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.exchangeRateMantissa}), redeemTokensIn
            );
            if (vars.mathErr != MathError.NO_ERROR) {
                return failOpaque(
                    Error.MATH_ERROR,
                    FailureInfo.REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED,
                    uint(vars.mathErr)
                );
            }
        } 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 : vars.exchangeRateMantissa})
                );
                if (vars.mathErr != MathError.NO_ERROR) {
                    return failOpaque(
                        Error.MATH_ERROR,
                        FailureInfo.REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED,
                        uint(vars.mathErr)
                    );
                }

                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, allowed);
            }

            /* 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_FAILED,
                    uint(vars.mathErr)
                );
            }

            (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_FAILED,
                    uint(vars.mathErr)
                );
            }

            /* 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
occurred.
              */
            uint _amount = vars.redeemAmount;
            uint _feeAmount = div_(mul_(_amount, redeemFee), expScale);
            totalReserves = add_(totalReserves, _feeAmount);
            _amount = sub_(_amount, _feeAmount);
            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) {
        address riskControl = Comptroller(address(comptroller)).getRiskControl();
        if (riskControl != address(0)) {
            uint risk = RiskController(riskControl).checkMintRisk(address(this), msg.sender);
            require(risk == 0, 'risk control');
        }
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted
borrow failed
            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, allowed);
        }

        /* 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_FAILED,
                uint(vars.mathErr)
            );
        }

        (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_CALCULATION_FAILED,
                uint(vars.mathErr)
            );
        }

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

```
        }

        ////////////////////////
        // 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 occurred.
          */
        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),
      *             and the actual repayment amount.
      */
    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 attempted borrow failed
            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),
      *     and the actual repayment amount.
      */
    function repayBorrowBehalfInternal(address borrower, 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 attempted borrow failed
            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),
      *     and the actual repayment amount.
      */
    function repayBorrowFresh(address payer, address borrower, uint repayAmount) internal returns
```

```
(uint, uint) {
        /* 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_REJECTION,
                allowed
            ), 0);
        }

        /* 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_CALCULATION_FAILED,
                uint(vars.mathErr)
            ), 0);
        }

        /* 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 occurred.
          *   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_CALCULATION_FAILED");

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

        /* 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.totalBorrowsNew);

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

        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),
     *                  and the actual repayment amount.
     */

    function liquidateBorrowInternal(
        address borrower,
        uint repayAmount,
        CTokenInterface cTokenCollateral
    ) 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 attempted liquidation failed
            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 attempted liquidation failed
            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),
     *                  and the actual repayment amount.
     */
    function liquidateBorrowFresh(
        address liquidator,
        address borrower,
        uint repayAmount,
        CTokenInterface cTokenCollateral
    ) internal returns (uint, uint) {
        /* Fail if liquidate not allowed */
        uint allowed = comptroller.liquidateBorrowAllowed(
            address(this),
            address(cTokenCollateral),
            liquidator,
            borrower,
            repayAmount
        );
        if (allowed != 0) {
            return (failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.LIQUIDATE_COMPTROLLER_REJECTION, allowed), 0);
        }
        /* 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_ZERO), 0);
        }

        /* Fail if repayAmount = -1 */
```

```
            if (repayAmount == uint(-1)) {
                return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED, FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX), 0);
            }

            /* Fail if repayBorrow fails */
            (uint repayBorrowError, uint actualRepayAmount) = repayBorrowFresh(liquidator, borrower, repayAmount);
            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(this),
                address(cTokenCollateral),
                actualRepayAmount
            );
            require(amountSeizeError == uint(Error.NO_ERROR), "LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED");

            /* 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 external call
            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), seizeTokens);

            /* We call the defense hook */
            comptroller.liquidateBorrowVerify(
                address(this),
                address(cTokenCollateral),
                liquidator,
                borrower,
                actualRepayAmount,
                seizeTokens
            );

            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 returns (uint) {
            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 of another CToken.
         *  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 seizeTokens
        ) internal returns (uint) {
            /* Fail if seize not allowed */
            uint allowed = comptroller.seizeAllowed(address(this), seizerToken, liquidator, borrower, seizeTokens);
            if (allowed != 0) {
                return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.LIQUIDATE_SEIZE_COMPTROLLER_REJECTION, allowed);
            }

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

            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, uint(mathErr));
            }

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

            ////////////////////////
            // 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 finalize the transfer.
          * @dev Admin function to begin change of admin.
          *      The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
          * @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);
    }

    function _setFeeManager(address payable _feeManager) external returns (uint){
        require(msg.sender == admin, "!admin");
        feeManager = _feeManager;
        return 0;
    }

    function _setRedeemFactorMantissa(uint _redeemFee) external {
        require(msg.sender == admin, "!admin");
        require(_redeemFee > 0 && _redeemFee < 1e16, "beyond the scope");
        redeemFee = _redeemFee;

    }

    /**
        * @notice accrues interest and sets a new reserve factor for the protocol using _setReserveFactorFresh
        * @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 an attempted reserve factor change failed.
            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 an attempted reduce reserves failed.
            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 details))
     *         and the actual amount added, net token fees
     */
    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), actualAddAmount);
        }

        ////////////////////////
        // 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 occurred.
         *  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 an attempted reduce reserves failed.
            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 occurred.
        doTransferOut(feeManager, 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 an attempted change of interest rate model failed
            return fail(Error(error), FailureInfo.SET_INTEREST_RATE_MODEL_ACCRUE_INTEREST_FAILED);
        }
        // _setInterestRateModelFresh emits interest-rate-model-update-specific logs on errors, so we don't need to.
        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 (uint) {

        // 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 to the protocol, in case of a fee.
      *    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 than reverting.
      *    If caller has not called checked protocol's balance, may revert due to insufficient cash held in the contract.
      *    If caller has checked protocol's balance, and verified it is >= amount,
      *    this should not revert in normal conditions.
      */
    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
    }


    /**
  * @dev allows smartcontracts to access the liquidity of the pool within one transaction,
  * as long as the amount taken plus a fee is returned.
                NOTE There are security concerns for developers of flashloan receiver contracts
  * that must be kept into consideration. For further details please visit https://developers.aave.com
  * @param _receiver The address of the contract receiving the funds.
            The receiver should implement the IFlashLoanReceiver interface.
  * @param _reserve the address of the principal reserve
  * @param _amount the amount requested for this flashloan
  **/
    function flashLoan(address _receiver, address _reserve, uint256 _amount, bytes memory _params)
    public nonReentrant returns (uint)
    {
        //check that the reserve has enough available liquidity
        //we avoid using the getAvailableLiquidity() function in LendingPoolCore to save gas
        uint256 availableLiquidityBefore = getCashPrior();

        require(
            availableLiquidityBefore >= _amount,
            "There is not enough liquidity available to borrow"
        );

        //calculate amount fee
        uint256 amountFee = div_(mul_(_amount, FLASHLOAN_FEE_TOTAL), 10000);

        require(
            amountFee > 0,
            "The requested amount is too small for a flashLoan."
        );

        //get the FlashLoanReceiver instance
        IFlashLoanReceiver receiver = IFlashLoanReceiver(_receiver);

        address payable userPayable = address(uint160(_receiver));

        //transfer funds to the receiver
        doTransferOut(userPayable, _amount);

        //execute action of the receiver
        receiver.executeOperation(_reserve, address(this), _amount, amountFee, _params);

        //check that the actual balance of the core contract includes the returned amount
        uint256 availableLiquidityAfter = getCashPrior();

        require(
            availableLiquidityAfter >= add_(availableLiquidityBefore, amountFee),
            "The actual balance of the protocol is inconsistent"
        );

        totalReserves = add_(totalReserves, amountFee);

        //solium-disable-next-line
        emit FlashLoan(_receiver, address(this), _reserve, _amount, amountFee, block.timestamp);
        return 0;
    }
}
```

***ErrorReporter.sol***

```
pragma solidity ^0.5.9;

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` is an arbitrary
      * contract-specific code that enables us to report opaque error codes from upgradeable contracts.
      **/
    event Failure(uint error, uint info, uint detail);

    /**
      * @dev use this when reporting a known error from the money market or a non-upgradeable collaborator
      */
    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,
        RISK_CONTROL
    }

    /*
     * 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` is an arbitrary
     * contract-specific code that enables us to report opaque error codes from upgradeable
```

```
contracts.
     **/
    event Failure(uint error, uint info, uint detail);

    /**
      * @dev use this when reporting a known error from the money market or a non-upgradeable collaborator
      */
    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);
    }
}
```

***JumpRateModel.sol***

```
pragma solidity ^0.5.9;

import "./interface/InterestRateModel.sol";
import "./lib/SafeMath.sol";

/**
  * @title Compound's JumpRateModel Contract
  * @author Compound
  */
contract JumpRateModel is InterestRateModel {
    using SafeMath for uint;

    event NewInterestParams(uint baseRatePerBlock, uint multiplierPerBlock, uint jumpMultiplierPerBlock, uint kink);

    /**
      * @notice The approximate number of blocks per year that is assumed by the interest rate model
      */
    uint public constant blocksPerYear = 10512000;

    /**
      * @notice The multiplier of utilization rate that gives the slope of the interest rate
      */
    uint public multiplierPerBlock;

    /**
      * @notice The base interest rate which is the y-intercept when utilization rate is 0
      */
    uint public baseRatePerBlock;

    /**
      * @notice The multiplierPerBlock after hitting a specified utilization point
      */
    uint public jumpMultiplierPerBlock;

    /**
      * @notice The utilization point at which the jump multiplier is applied
      */
    uint public kink;

    /**
      * @notice Construct an interest rate model
      * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
      * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
      * @param jumpMultiplierPerYear The multiplierPerBlock after hitting a specified utilization point
      * @param kink_ The utilization point at which the jump multiplier is applied
      */
    constructor(uint baseRatePerYear, uint multiplierPerYear, uint jumpMultiplierPerYear, uint kink_) public {
        baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
        multiplierPerBlock = multiplierPerYear.div(blocksPerYear);
        jumpMultiplierPerBlock = jumpMultiplierPerYear.div(blocksPerYear);
        kink = kink_;

        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock, jumpMultiplierPerBlock, kink);
    }

    /**
      * @notice Calculates the utilization rate of the market: `borrows / (cash + borrows - reserves)`
```

```
        * @param cash The amount of cash in the market
        * @param borrows The amount of borrows in the market
        * @param reserves The amount of reserves in the market (currently unused)
        * @return The utilization rate as a mantissa between [0, 1e18]
        */
    function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }

        return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
    }

    /**
      * @notice Calculates the current borrow rate per block, with the error code expected by the market
      * @param cash The amount of cash in the market
      * @param borrows The amount of borrows in the market
      * @param reserves The amount of reserves in the market
      * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
      */
    function getBorrowRate(uint cash, uint borrows, uint reserves) public view returns (uint) {
        uint util = utilizationRate(cash, borrows, reserves);

        if (util <= kink) {
            return util.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
        } else {
            uint normalRate = kink.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
            uint excessUtil = util.sub(kink);
            return excessUtil.mul(jumpMultiplierPerBlock).div(1e18).add(normalRate);
        }
    }

    /**
      * @notice Calculates the current supply rate per block
      * @param cash The amount of cash in the market
      * @param borrows The amount of borrows in the market
      * @param reserves The amount of reserves in the market
      * @param reserveFactorMantissa The current reserve factor for the market
      * @return The supply rate percentage per block as a mantissa (scaled by 1e18)
      */
    function getSupplyRate(
        uint cash,
        uint borrows,
        uint reserves,
        uint reserveFactorMantissa
    ) public view returns (uint) {
        uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
        uint borrowRate = getBorrowRate(cash, borrows, reserves);
        uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
        return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
    }
}
```

***JumpRateModelV2.sol***

```
pragma solidity ^0.5.9;

import "./BaseJumpRateModelV2.sol";
import "./interface/InterestRateModel.sol";


/**
  * @title Compound's JumpRateModel Contract V2 for V2 cTokens
  * @author Arr00
  * @notice Supports only for V2 cTokens
  */
contract JumpRateModelV2 is InterestRateModel, BaseJumpRateModelV2 {

    /**
      * @notice Calculates the current borrow rate per block
      * @param cash The amount of cash in the market
      * @param borrows The amount of borrows in the market
      * @param reserves The amount of reserves in the market
      * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
      */
    function getBorrowRate(uint cash, uint borrows, uint reserves) external view returns (uint) {
        return getBorrowRateInternal(cash, borrows, reserves);
    }

    constructor(uint baseRatePerYear, uint multiplierPerYear, uint jumpMultiplierPerYear, uint kink_, address owner_)
    BaseJumpRateModelV2(baseRatePerYear, multiplierPerYear, jumpMultiplierPerYear, kink_, owner_) public {}
}
```

***LegacyJumpRateModelV2.sol***

```
pragma solidity ^0.5.9;

import "./BaseJumpRateModelV2.sol";
import "./interface/LegacyInterestRateModel.sol";


/**
  * @title Compound's JumpRateModel Contract V2 for legacy cTokens
  * @author Arr00
  * @notice Supports only legacy cTokens
  */
contract LegacyJumpRateModelV2 is LegacyInterestRateModel, BaseJumpRateModelV2 {

    /**
     * @notice Calculates the current borrow rate per block, with the error code expected by the market
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market
     * @return (Error, The borrow rate percentage per block as a mantissa (scaled by 1e18))
     */
    function getBorrowRate(uint cash, uint borrows, uint reserves) external view returns (uint, uint) {
        return (0, getBorrowRateInternal(cash, borrows, reserves));
    }

    constructor(uint baseRatePerYear, uint multiplierPerYear, uint jumpMultiplierPerYear, uint kink_, address owner_)
    BaseJumpRateModelV2(baseRatePerYear, multiplierPerYear, jumpMultiplierPerYear, kink_, owner_) public {}
}
```

***Maximillion.sol***

```
pragma solidity ^0.5.9;

import "./CNative.sol";

/**
 * @title Compound's Maximillion Contract
 * @author Compound
 */
contract Maximillion {
    /**
     * @notice The default cEther market to repay in
     */
    CNative public cEther;

    /**
     * @notice Construct a Maximillion to repay max in a CEther market
     */
    constructor(CNative cEther_) public {
        cEther = cEther_;
    }

    /**
     * @notice msg.sender sends Ether to repay an account's borrow in the cEther market
     * @dev The provided Ether is applied towards the borrow balance, any excess is refunded
     * @param borrower The address of the borrower account to repay on behalf of
     */
    function repayBehalf(address borrower) public payable {
        repayBehalfExplicit(borrower, cEther);
    }

    /**
     * @notice msg.sender sends Ether to repay an account's borrow in a cEther market
     * @dev The provided Ether is applied towards the borrow balance, any excess is refunded
     * @param borrower The address of the borrower account to repay on behalf of
     * @param cEther_ The address of the cEther contract to repay in
     */
    function repayBehalfExplicit(address borrower, CNative cEther_) public payable {
        uint received = msg.value;
        uint borrows = cEther_.borrowBalanceCurrent(borrower);
        if (received > borrows) {
            cEther_.repayBorrowBehalf.value(borrows)(borrower);
            msg.sender.transfer(received - borrows);
        } else {
            cEther_.repayBorrowBehalf.value(received)(borrower);
        }
    }
}
```

***Migrations.sol***

```
pragma solidity ^0.5.9;

contract Migrations {

}
```

***RiskController.sol***

```
pragma solidity ^0.5.9;

import "./lib/Controller.sol";

contract RiskController is Controller {

    uint constant   NO_RISK = 0;
    uint constant   HIGH = 1;


    mapping(address => bool) public contractWhiteList;

    mapping(address => bool) public banMint;
    mapping(address => bool) public banBorrow;

    function isContract(address _addr) public view returns (bool isContract){
        uint32 size;
        assembly {
            size := extcodesize(_addr)
        }
        return (size > 0);
    }


    function addToWhiteList(address _target) public onlyController {
        contractWhiteList[_target] = true;
    }

    function removeFromWhiteList(address _target) public onlyController {
        contractWhiteList[_target] = false;
    }

    function banTokenMint(address _token) public onlyController {
        banMint[_token] = true;
    }

    function removeBanTokenMint(address _token) public onlyController {
        banMint[_token] = false;
    }

    function banTokenBorrow(address _token) public onlyController {
        banBorrow[_token] = true;
    }

    function removeBanTokenBorrow(address _token) public onlyController {
        banBorrow[_token] = false;
    }

    function checkMintRisk(
        address _token,
        address _address
    ) external view returns (uint mintRisk) {
        if (!isContract(_address)) {
            return NO_RISK;
        }
        if (contractWhiteList[_address]) {
            return NO_RISK;
        }
        mintRisk = banMint[_token] ? HIGH : NO_RISK;
        return mintRisk;

    }

    function checkBorrowRisk(
        address _token,
        address _address
    ) external view returns (uint borrowRisk) {
        if (!isContract(_address)) {
            return NO_RISK;
        }
        if (contractWhiteList[_address]) {
            return NO_RISK;
        }
        borrowRisk = banBorrow[_token] ? HIGH : NO_RISK;
        return borrowRisk;
    }

}
```

**Unitroller.sol**

```
pragma solidity ^0.5.9;

import "./ErrorReporter.sol";
import "./interface/ComptrollerStorage.sol";
/**
  * @title ComptrollerCore
  * @dev Storage for the comptroller is at this address, while execution is delegated to the `comptrollerImplementation`.
  * 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 implementation is updated
      */
    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 == address(0)) {
            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 finalize the transfer.
      * @dev Admin function to begin change of admin.
      *      The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
      * @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)}
        }
    }
}
```

***WhitePaperInterestRateModel.sol***

```
pragma solidity ^0.5.9;

import "./interface/InterestRateModel.sol";
import "./lib/SafeMath.sol";

/**
  * @title Compound's WhitePaperInterestRateModel Contract
  * @author Compound
```

```
    * @notice The parameterized model described in section 2.4 of the original Compound Protocol whitepaper
    */
contract WhitePaperInterestRateModel is InterestRateModel {
    using SafeMath for uint;

    event NewInterestParams(uint baseRatePerBlock, uint multiplierPerBlock);

    /**
     * @notice The approximate number of blocks per year that is assumed by the interest rate model
     */
    uint public constant blocksPerYear = 10512000;

    /**
     * @notice The multiplier of utilization rate that gives the slope of the interest rate
     */
    uint public multiplierPerBlock;

    /**
     * @notice The base interest rate which is the y-intercept when utilization rate is 0
     */
    uint public baseRatePerBlock;

    /**
     * @notice Construct an interest rate model
     * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
     * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
     */
    constructor(uint baseRatePerYear, uint multiplierPerYear) public {
        baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
        multiplierPerBlock = multiplierPerYear.div(blocksPerYear);

        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock);
    }

    /**
     * @notice Calculates the utilization rate of the market: `borrows / (cash + borrows - reserves)`
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market (currently unused)
     * @return The utilization rate as a mantissa between [0, 1e18]
     */
    function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }

        return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
    }

    /**
     * @notice Calculates the current borrow rate per block, with the error code expected by the market
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market
     * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
     */
    function getBorrowRate(uint cash, uint borrows, uint reserves) public view returns (uint) {
        uint ur = utilizationRate(cash, borrows, reserves);
        return ur.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
    }

    /**
     * @notice Calculates the current supply rate per block
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market
     * @param reserveFactorMantissa The current reserve factor for the market
     * @return The supply rate percentage per block as a mantissa (scaled by 1e18)
     */
    function getSupplyRate(
        uint cash,
        uint borrows,
        uint reserves,
        uint reserveFactorMantissa
    ) public view returns (uint) {
        uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
        uint borrowRate = getBorrowRate(cash, borrows, reserves);
        uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
        return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
    }
}
```

# 6. 附录 B：安全风险评级标准

| *智能合约漏洞评级标准* | |
|---|---|
| **漏洞评级** | **漏洞评级说明** |
| 高危漏洞 | 能直接造成代币合约或用户资金损失的漏洞，如：能造成代币价值归零的数值溢出漏洞、能造成交易所损失代币的假充值漏洞、能造成合约账户损失 TRX 或代币的重入漏洞等；<br>能造成代币合约归属权丢失的漏洞，如：关键函数的访问控制缺陷、call 注入导致关键函数访问控制绕过等；<br>能造成代币合约无法正常工作的漏洞，如：因向恶意地址发送 TRX 导致的拒绝服务漏洞、因 energy 耗尽导致的拒绝服务漏洞。 |
| 中危漏洞 | 需要特定地址才能触发的高风险漏洞，如代币合约拥有者才能触发的数值溢出漏洞等；非关键函数的访问控制缺陷、不能造成直接资金损失的逻辑设计缺陷等。 |
| 低危漏洞 | 难以被触发的漏洞、触发之后危害有限的漏洞，如需要大量 TRX 或代币才能触发的数值溢出漏洞、触发数值溢出后攻击者无法直接获利的漏洞、通过指定高 energy 触发的事务顺序依赖风险等。 |

# 7. 附录 C：智能合约安全审计工具简介

## 7.1 Manticore

Manticore 是一个分析二进制文件和智能合约的符号执行工具，Manticore 包含一个符号虚拟机（EVM），一个 EVM 反汇编器/汇编器以及一个用于自动编译和分析 Solidity 的方便界面。它还集成了 Ethersplay，用于 EVM 字节码的 Bit of Traits of Bits 可视化反汇编程序，用于可视化分析。与二进制文件一样，Manticore 提供了一个简单的命令行界面和一个用于分析 EVM 字节码的 Python API。

## 7.2 Oyente

Oyente 是一个智能合约分析工具，Oyente 可以用来检测智能合约中常见的 bug，比如 reentrancy、事务排序依赖等等。更方便的是，Oyente 的设计是模块化的，所以这让高级用户可以实现并插入他们自己的检测逻辑，以检查他们的合约中自定义的属性。

## 7.3 securify.sh

Securify 可以验证智能合约常见的安全问题，例如交易乱序和缺少输入验证，它在全自动化的同时分析程序所有可能的执行路径，此外，Securify 还具有用于指定漏洞的特定语言，这使 Securify 能够随时关注当前的安全性和其他可靠性问题。

## 7.4 Echidna

Echidna 是一个为了对 EVM 代码进行模糊测试而设计的 Haskell 库。

## 7.5 MAIAN

MAIAN 是一个用于查找智能合约漏洞的自动化工具，Maian 处理合约的字

节码，并尝试建立一系列交易以找出并确认错误。

## 7.6 ethersplay

ethersplay 是一个 EVM 反汇编器，其中包含了相关分析工具。

## 7.7 ida-evm

ida-evm 是一个针对虚拟机（EVM）的 IDA 处理器模块。

## 7.8 Remix-ide

Remix 是一款基于浏览器的编译器和 IDE，可让用户使用 Solidity 语言构建合约并调试交易。

## 7.9 知道创宇区块链安全审计人员专用工具包

知道创宇安全审计人员专用工具包，由知道创宇渗透测试工程师研发，收集和使用，包含专用于测试人员的批量自动测试工具，自主研发的工具、脚本或利用工具等。

知道创宇

北京知道创宇信息技术股份有限公司


咨询电话 +86(10)400 060 9587

邮　　箱 sec@knownsec.com

官　　网 www.knownsec.com

地　　址 北京市 朝阳区 望京 SOHO T2-B座-2509