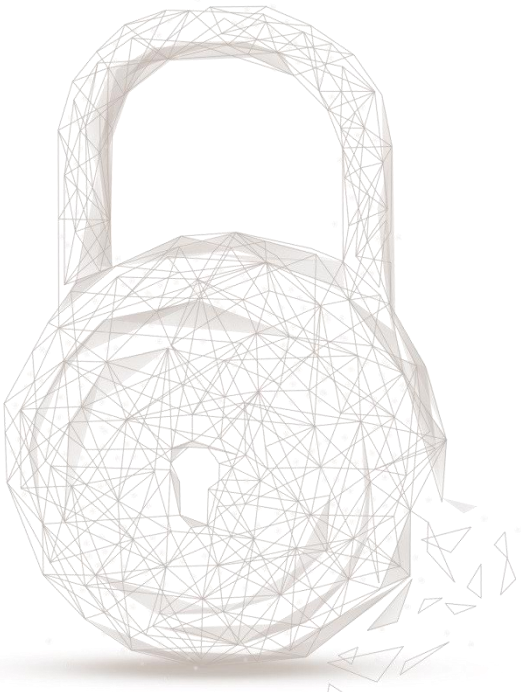
成都链安
BEOSIN

智能合约安全审计报告



成都链安 BEOSIN

审计编号：202103261342

报告查询名称：Goswap

审计合约名称	审计合约地址	审计合约链接
MasterChef	0x7dCeBC34F55b52df742C91581089ebD0BCBD254F	https://hecoinfo.com/address/0x7dCeBC34F55b52df742C91581089ebD0BCBD254F#code
LPTokenPool	0xC33F68fBBCB529faB10aB5FcFD77BaD7cE9fbfFA	https://hecoinfo.com/address/0xC33F68fBBCB529faB10aB5FcFD77BaD7cE9fbfFA#code

合约审计开始日期：2021.03.22

合约审计完成日期：2021.03.26

审计结果：通过

审计团队：成都链安科技有限公司

审计类型及结果：

序号	审计类型	审计子项	审计结果
1	代码规范审计	编译器版本安全审计	通过
		弃用项审计	通过
		冗余代码审计	通过
		require/assert 使用审计	通过
		gas 消耗审计	通过
2	通用漏洞审计	整型溢出审计	通过
		重入攻击审计	通过
		伪随机数生成审计	通过
		交易顺序依赖审计	通过
		拒绝服务攻击审计	通过
		函数调用权限审计	通过
		call/delegatecall 安全审计	通过

		返回值安全审计	通过
		tx.origin 使用安全审计	通过
		重放攻击审计	通过
		变量覆盖审计	通过
3	业务审计	业务逻辑审计	通过
		业务实现审计	通过

免责声明：本次审计仅针对本报告载明的审计类型及结果表中给定的审计类型范围进行审计，其他未知安全漏洞不在本次审计责任范围之内。成都链安科技仅根据本报告出具前已经存在或发生的攻击或漏洞出具本报告，对于出具以后存在或发生的新的攻击或漏洞，成都链安科技无法判断其对智能合约安全状况可能的影响，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，仅基于合约提供者在本报告出具前已向成都链安科技提供的文件和资料，且该部分文件和资料不存在任何缺失，被篡改，删减或隐瞒的前提下作出的；如提供的文件和资料存在信息缺失，被篡改，删减，隐瞒或反映的情况与实际情况不符等情况或提供文件和资料在本报告出具后发生任何变动的，成都链安科技对由此而导致的损失和不利影响不承担任何责任。成都链安科技出具的本审计报告系根据合约提供者提供的文件和资料依靠成都链安科技现掌握的技术而作出的，由于任何机构均存在技术的局限性，成都链安科技作出的本审计报告仍存在无法完整检测出全部风险的可能性，成都链安科技对由此产生的损失不承担任何责任。

本声明最终解释权归成都链安科技所有。

审计结果说明：

本公司采用形式化验证，静态分析，动态分析，典型案例测试和人工审核的方式对Goswap项目智能合约代码规范性，安全性以及业务逻辑三个方面进行多维度全面的安全审计。**经审计，Goswap项目智能合约通过所有检测项，合约审计结果为通过。**以下为本合约详细审计信息。

一. 代码规范审计

1. 编译器版本安全审计

老版本的编译器可能会导致各种已知的安全问题，建议开发者在代码中指定合约代码采用最新的编译器版本，并消除编译器告警。

- **安全建议：无**
- **审计结果：通过**

2. 弃用项审计

Solidity智能合约开发语言处于快速迭代中，部分关键字已被新版本的编译器弃用，如throw，years等，为了消除其可能导致的隐患，合约开发者不应该使用当前编译器版本已弃用的关键字。

- **安全建议：无**
- **审计结果：通过**

3. 冗余代码审计

智能合约中的冗余代码会降低代码可读性，并可能需要消耗更多的gas用于合约部署，建议消除冗余代码。

- **安全建议：无**
- **审计结果：通过**

4. require/assert 使用审计

Solidity使用状态恢复异常来处理错误。这种机制将会撤消对当前调用(及其所有子调用)中的状态所做的所有更改，并向调用者标记错误。函数assert和require可用于检查条件并在条件不满足时抛出异常。assert函数只能用于测试内部错误，并检查非变量。require函数用于确认条件有效性，例如输入变量，或合约状态变量是否满足条件，或验证外部合约调用的返回值。

- **安全建议：无**
- **审计结果：通过**

5. gas 消耗审计

Heco虚拟机执行合约代码需要消耗gas，当gas不足时，代码执行会抛出out of gas异常，并撤销所有状态变更。合约开发者需要控制代码的gas消耗，避免因为gas不足导致函数执行一直失败。

- **安全建议：无**
- **审计结果：通过**

二. 通用漏洞审计

1. 整型溢出审计

整型溢出是很多语言都存在的安全问题，它们在智能合约中尤其危险。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uint类型时，0减去1会下溢得到最大数字值。溢出情况会导致不正确的结果，特别是如果其可能的结果未被预期，可能会影响程序的可靠性和安全性。

- **安全建议：无**
- **审计结果：通过**

2. 重入攻击审计

重入漏洞是最典型的智能合约漏洞，该漏洞原因是Solidity中的call.value()函数在被用来发送HT的时候会消耗它接收到的所有gas，当调用call.value()函数发送HT的逻辑顺序存在错误时，就会存在重入攻击的风险。

- **安全建议：无**
- **审计结果：通过**

3. 伪随机数生成审计

智能合约中可能会使用到随机数，在solidity下常见的是用block区块信息作为随机因子生成，但是这样使用是不安全的，区块信息是可以被矿工控制或被攻击者在交易时获取到，这类随机数在一定程度上是可预测或可碰撞的，比较典型的例子就是fomo3d的airdrop随机数可以被碰撞。

- **安全建议：无**
- **审计结果：通过**

4. 交易顺序依赖审计

在Heco的交易打包执行过程中，面对相同难度的交易时，矿工往往会选择gas费用高的优先打包，因此用户可以指定更高的gas费用，使自己的交易优先被打包执行。

- **安全建议：无**
- **审计结果：通过**

5. 拒绝服务攻击审计

拒绝服务攻击，即Denial of Service，可以使目标无法提供正常的服务。在Heco智能合约中也会存在此类问题，由于智能合约的不可更改性，该类攻击可能使得合约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有很多种，包括在作为交易接收方时的恶意revert，代码设计缺陷导致gas耗尽等等。

- **安全建议：无**
- **审计结果：通过**

6. 函数调用权限审计

智能合约如果存在高权限功能，如：铸币，自毁，change owner等，需要对函数调用做权限限制，避免权限泄露导致的安全问题。

- **安全建议：无**
- **审计结果：通过**

7. call/delegatecall安全审计

Solidity中提供了call/delegatecall函数来进行函数调用，如果使用不当，会造成call注入漏洞，例如call的参数如果可控，则可以控制本合约进行越权操作或调用其他合约的危险函数。

- **安全建议：无**

- **审计结果：通过**

8. 返回值安全审计

在Solidity中存在transfer()，send()，call.value()等方法中，transfer转账失败交易会回滚，而send和call.value转账失败会return false，如果未对返回做正确判断，则可能会执行到未预期的逻辑；另外在TRC20 Token的transfer/transferFrom功能实现中，也要避免转账失败return false的情况，以免造成假充值漏洞。

- **安全建议：无**
- **审计结果：通过**

9. tx.origin使用安全审计

在Heco智能合约的复杂调用中，tx.origin表示交易的初始创建者地址，如果使用tx.origin进行权限判断，可能会出现错误；另外，如果合约需要判断调用方是否为合约地址时则需要使用tx.origin，不能使用extcodesize。

- **安全建议：无**
- **审计结果：通过**

10. 重放攻击审计

重放攻击是指如果两份合约使用了相同的代码实现，并且身份鉴权在传参中，当用户在向一份合约中执行一笔交易，交易信息可以被复制并且向另一份合约重放执行该笔交易。

- **安全建议：无**
- **审计结果：通过**

11. 变量覆盖审计

Heco存在着复杂的变量类型，例如结构体，动态数组等，如果使用不当，对其赋值后，可能导致覆盖已有状态变量的值，造成合约执行逻辑异常。

- **安全建议：无**
- **审计结果：通过**

三. 业务审计

3.1 LPTokenPool 合约审计

(1) 抵押奖励参数设置

- **业务描述：**合约的“抵押-奖励”模式需要设置相关奖励计算参数（奖励比例rewardRate、上次更新时间lastUpdateTime、阶段完成时间periodFinish）。通过指定的奖励分配管理员地址rewardDistribution调用notifyRewardAmount函数，可设置抵押奖励参数。“抵押-奖励”模式启动之后，rewardDistribution可指定增加的奖励数值reward，用于更新奖励比例rewardRate。

```
    function notifyRewardAmount(uint256 reward)
        external
        override
        onlyRewardDistribution
        updateReward(address(0))
    {
        // 如果当前时间>开始时间
        if (block.timestamp > starttime) {
            // 如果当前时间 >= 结束时间
            if (block.timestamp >= periodFinish) {
                // 每秒奖励 = 奖励数量 / 180天
                rewardRate = reward.div(DURATION);
            } else {
                // 剩余时间 = 结束时间 - 当前时间
                uint256 remaining = periodFinish.sub(block.timestamp);
                // 剩余奖励数量 = 剩余时间 * 每秒奖励 (第一次执行为0)
                uint256 leftover = remaining.mul(rewardRate);
                // 每秒奖励 = (奖励数量 + 剩余奖励数量) / 180天
                rewardRate = reward.add(leftover).div(DURATION);
            }
            //最后更新时间 = 当前时间
            lastUpdateTime = block.timestamp;
            // 结束时间 = 当前时间 + 180天
            periodFinish = block.timestamp.add(DURATION);
            // 触发奖励增加事件
            emit RewardAdded(reward);
        } else {
            // 每秒奖励 = 奖励数量 / 180天
            rewardRate = reward.div(DURATION);
            // 最后更新时间 = 开始时间
            lastUpdateTime = starttime;
            // 结束时间 = 开始时间 + 180天
            periodFinish = starttime.add(DURATION);
            // 触发奖励增加事件
            emit RewardAdded(reward);
        }
    }
}
```

图 1 notifyRewardAmount 函数源码截图

- **相关函数：**notifyRewardAmount、rewardPerToken、lastTimeRewardApplicable
- **安全建议：无**
- **审计结果：通过**

(2) 抵押代币

- **业务描述：**合约实现了stake函数用于抵押lpt代币，用户预先授权该合约地址，通过调用lpt合约中的transferFrom函数，合约地址代理用户将指定数量的lpt代币转至本合约地址；该函数限制用户仅可在“抵押-奖励”模式开启（block.timestamp >= starttime）后进行调用；每次调用该函数抵押代币时通过修饰器updateReward更新奖励相关数据。

```
function stake(uint256 amount)
    public
    override
    updateReward(msg.sender)
    checkStart
{
    // 确认数量>0
    require(amount > 0, 'HUSDGOCLPTokenSharePool: Cannot stake 0');
    // 上级质押
    super.stake(amount);
    // 触发质押事件
    emit Staked(msg.sender, amount);
}
```

图 2 stake 函数源码截图（1/2）

```
function stake(uint256 amount) public virtual {
    // 总量增加
    _totalSupply = _totalSupply.add(amount);
    // 余额映射增加
    _balances[msg.sender] = _balances[msg.sender].add(amount);
    // 将LPToken发送到当前合约
    lpt.safeTransferFrom(msg.sender, address(this), amount);
}
```

图 3 stake 函数源码截图（2/2）

```
 */
modifier checkStart() {
    require(block.timestamp >= starttime, 'LPTokenSharePool: not start');
    _;
}
```

图 4 checkStart 修饰器源码截图

- **相关函数：**stake、rewardPerToken、lastTimeRewardApplicable、earned、balanceOf
- **安全建议：无**
- **审计结果：通过**

(3) 提取抵押代币

- **业务描述：**合约实现了withdraw函数用于提取已抵押的lpt代币，通过调用lpt合约中的transfer函数，合约地址将指定数量的lpt代币转至函数调用者（用户）地址；该函数限制用户仅可在“抵押-奖励”模式开启（block.timestamp >= starttime）后进行调用；每次调用该函数抵押代币时通过修饰器updateReward更新奖励相关数据。

```
    */
    function withdraw(uint256 amount)
        public
        override
        updateReward(msg.sender)
        checkStart
    {
        // 确认数量>0
        require(amount > 0, 'HUSDGOCLPTokenSharePool: Cannot withdraw 0');
        // 上级提款
        super.withdraw(amount);
        // 触发提款事件
        emit Withdrawn(msg.sender, amount);
    }
```

图 5 withdraw 函数源码截图（1/2）

```
function withdraw(uint256 amount) public virtual {
    // 用户的总质押数量
    uint256 directorShare = _balances[msg.sender];
    // 确认总质押数量大于取款数额
    require(directorShare >= amount, 'withdraw request greater than staked amount');
    // 总量减少
    _totalSupply = _totalSupply.sub(amount);
    // 余额减少
    _balances[msg.sender] = _balances[msg.sender].sub(amount);
    // 将LPToken发送给用户
    lpt.safeTransfer(msg.sender, amount);
}
```

图 6 withdraw 函数源码截图（2/2）

- **相关函数：**withdraw、rewardPerToken、lastTimeRewardApplicable、earned、balanceOf
- **安全建议：无**
- **审计结果：通过**

(4) 领取抵押奖励

- **业务描述：**合约实现了getReward函数用于领取抵押奖励，通过调用合约中的transfer函数，合约地址将指定数量（用户的全部抵押奖励）的代币转至函数调用者（用户）地址；该函数限制用户仅可在“抵押-奖励”模式开启（block.timestamp >= starttime）后进行调用；每次调用该函数抵押代币时通过修饰器updateReward更新奖励相关数据。

```
function getReward() public updateReward(msg.sender) checkStart {
    // 奖励数量 = 用户已奖励的数量
    uint256 reward = earned(msg.sender);
    // 如果奖励数量>0
    if (reward > 0) {
        // 用户未发放的奖励数量 = 0
        rewards[msg.sender] = 0;
        // 发送奖励
        token.safeTransfer(msg.sender, reward);
        // 触发支付奖励事件
        emit RewardPaid(msg.sender, reward);
    }
}
```

图 7 getReward 函数源码截图

- **相关函数：**getReward、rewardPerToken、lastTimeRewardApplicable、earned、balanceOf
- **安全建议：无**
- **审计结果：通过**

(5) 提取所有代币并领取奖励

- **业务描述：**合约实现了exit函数用于调用者提取所有代币并领取奖励，调用withdraw函数提取全部已抵押的lpt代币，调用getReward函数领取调用者的抵押奖励，结束“抵押-奖励”模式参与。

```
function exit() external {
    // 提走用户质押的全部数量
    withdraw(balanceOf(msg.sender));
    // 获取奖励
    getReward();
}
```

图 8 exit 函数源码截图

- **相关函数：**exit、withdraw、getReward、rewardPerToken、lastTimeRewardApplicable、earned、balanceOf
- **安全建议：无**
- **审计结果：通过**

(6) 奖励相关数据查询功能

- **业务描述：**合约用户可通过调用lastTimeRewardApplicable函数查询当前时间戳与阶段完成时间中最早的时间戳；调用rewardPerToken函数可查询每个抵押代币可获得的抵押奖励；调用earned函数可查询指定地址所获取的总抵押奖励。

```
function lastTimeRewardApplicable() public view returns (uint256) {
    // 最小值(当前时间,结束时间)
    return Math.min(block.timestamp, periodFinish);
}

/**
 * @dev 每个质押Token的奖励
 * @return 奖励数量
 */
function rewardPerToken() public view returns (uint256) {
    // 返回0
    if (totalSupply() == 0) {
        return rewardPerTokenStored;
    }
    // 已奖励数量 + (min(当前时间,最后时间) - 最后更新时间) * 每秒奖励 * 1e18 / 质押总量
    return
        rewardPerTokenStored.add(
            lastTimeRewardApplicable()
                .sub(lastUpdateTime)
                .mul(rewardRate)
                .mul(1e18)
                .div(totalSupply())
        );
}

/**
 * @dev 用户已奖励的数量
 * @param account 用户地址
 */
function earned(address account) public view returns (uint256) {
    // 用户的质押数量 * (每个质押Token的奖励 - 每个质押Token支付用户的奖励) / 1e18 + 用户未发放的奖励数量
    return
        balanceOf(account)
            .mul(rewardPerToken().sub(userRewardPerTokenPaid[account]))
            .div(1e18)
            .add(rewards[account]);
}
```

图 9 其他相关函数源码截图

- **相关函数：**lastTimeRewardApplicable、rewardPerToken、earned
- **安全建议：无**
- **审计结果：通过**

3.2 MasterChef 合约审计

(1) constructor构造函数

- **业务描述：**构造函数用于设置奖励代币开始挖掘的区块号。

```
/**
 * @dev 构造函数
 * @param _startBlock GOT挖掘开始时的块号
 */
constructor(uint256 _startBlock) public {
    startBlock = _startBlock;
}
```

图 10 constructor 函数源码截图

- **安全建议：无**
- **审计结果：通过**

(2) 添加抵押池功能

- **业务描述：**如下图所示，合约实现了add函数用于添加LP代币的抵押池，仅合约的owner可以调用。添加新的抵押池时需要设置分配点数和LP代币的合约地址，选择设置是否需要先执行一次massUpdatePools函数更新所有抵押池。这里需要注意的是，不要重复添加同一个lptoken作为抵押，这样会导致计算奖励出错。

```
function add(
    uint256 _allocPoint,
    IERC20 _lpToken,
    bool _withUpdate
) public onlyOwner {
    // 触发更新所有池的奖励变量
    if (_withUpdate) {
        massUpdatePools();
    }
    // 分配发生的最后一个块号 = 当前块号 > GOT挖掘开始时的块号 > 当前块号 : GOT挖掘开始时的块号
    uint256 lastRewardBlock =
        block.number > startBlock ? block.number : startBlock;
    // 总分配点添加分配给该池的分配点数
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    // 池子信息推入池子数组
    poolInfo.push(
        PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accGOTPerShare: 0
        })
    );
}
```

图 11 add 函数源码截图

```
// Update reward variables for all pools. Be careful of g
function massUpdatePools() public {
    // 池子数量
    uint256 length = poolInfo.length;
    // 遍历所有池子
    for (uint256 pid = 0; pid < length; ++pid) {
        // 升级池子(池子id)
        updatePool(pid);
    }
}
```

图 12 massUpdatePools 函数源码截图

```
// Update reward variables of the given pool to be up-to-date.
function updatePool(uint256 _pid) public {
    // 实例化池子信息
    PoolInfo storage pool = poolInfo[_pid];
    // 如果当前区块号 <= 池子信息.分配发生的最后一个块号
    if (block.number <= pool.lastRewardBlock) {
        // 直接返回
        return;
    }
    // LPtoken的供应量 = 当前合约在`池子信息.lotoken地址`的余额
    uint256 lpSupply = pool.lpToken.balanceOf(address(this));
    // 如果 LPtoken的供应量 == 0
    if (lpSupply == 0) {
        // 池子信息.分配发生的最后一个块号 = 当前块号
        pool.lastRewardBlock = block.number;
        // 返回
        return;
    }
    // 奖金乘积 = 获取奖金乘积(分配发生的最后一个块号，当前块号)
    uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
    // GOT奖励 = 奖金乘积 * 每块创建的GOT令牌 * 池子分配点数 / 总分配点数
    uint256 GOTReward =
        multiplier.mul(GOTPerBlock).mul(pool.allocPoint).div(
            totalAllocPoint
        );
    // 开发者奖励为0.5%
    uint256 fundReserve = GOTReward.div(fundDivisor);
    // 调用GOT的铸造方法，为管理团队铸造 (`GOT奖励` / 20) token
    IGOT(GOT).mint(address(this), fundReserve);
    // 当前合约批准fund地址,开发者准备金数额
    IERC20(GOT).safeApprove(fund, fundReserve);
    // 调用fund合约的存款方法存入开发者准备金
    ISimpleERCFund(fund).deposit(
        GOT,
        fundReserve,
        "MasterChef: Fund Reserve"
    );
    // 调用GOT的铸造方法，为当前合约铸造 `GOT奖励` token
    IGOT(GOT).mint(address(this), GOTReward);
    // 每股累积GOT = 每股累积GOT + GOT奖励 * 1e12 / LPtoken的供应量
    pool.accGOTPerShare = pool.accGOTPerShare.add(
        GOTReward.mul(1e12).div(lpSupply)
    );
    // 池子信息.分配发生的最后一个块号 = 当前块号
    pool.lastRewardBlock = block.number;
}
```

图 13 updatePool 函数源码截图

- **相关函数：** add、massUpdatePools、updatePool、balanceOf、getMultiplier、mint、safeApprove
- **安全建议：无**
- **审计结果：通过**

(3) 更改指定抵押池参数功能

- **业务描述：**如下图所示，合约实现了set函数用于修改指定抵押池，仅合约owner可调用。可以修改抵押池的分配点数。可选择修改之前是否先执行一次massUpdatePools函数更新所有抵押池。

```
// Update the given pool's GOT allocation point. Can only be called by the owner.
function set(
    uint256 _pid,
    uint256 _allocPoint,
    bool _withUpdate
) public onlyOwner {
    // 触发更新所有池的奖励变量
    if (_withUpdate) {
        massUpdatePools();
    }
    // 总分配点 = 总分配点 - 池子数组[池子id].分配点数 + 新的分配给该池的分配点数
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
        _allocPoint
    );
    // 池子数组[池子id].分配点数 = 新的分配给该池的分配点数
    poolInfo[_pid].allocPoint = _allocPoint;
}
```

图 14 set 函数源码截图

- **相关函数：** set、massUpdatePools、updatePool、balanceOf、getMultiplier、mint、safeApprove
- **安全建议：无**
- **审计结果：通过**

(4) 设置迁移地址功能

- **业务描述：**如下图所示，合约实现了setMigrator函数用于设置迁移地址，仅合约owner可调用。

```
    */
    // Set the migrator contract. Can only be called by the owner.
    function setMigrator(IMigratorChef _migrator) public onlyOwner {
        migrator = _migrator;
    }
```

图 15 setMigrator 函数源码截图

- **安全建议：无**
- **审计结果：通过**

(5) 迁移功能

- **业务描述：**如下图所示，合约实现了migrate函数用于将指定pool池的所有抵押lp代币迁移到migrator地址，合同所有者调用setMigrator函数来设置migrator地址，任何用户都可以调用此函数来迁移指定池的所有抵押代币到migrator地址（要求migrator地址不等于零地址），该合约将指定池中的所有抵押lp代币授权给migrator地址，并调用migrator合约的migrate函数将所有抵押代币迁移（默认migrator为零地址）。（**注：如果migrator为恶意合约地址，用户的抵押代币存在丢失风险。**）

```
function migrate(uint256 _pid) public {
    // 确认迁移合约已经设置
    require(address(migrator) != address(0), "migrate: no migrator");
    // 实例化池子信息构造体
    PoolInfo storage pool = poolInfo[_pid];
    // 实例化LP token
    IERC20 lpToken = pool.lpToken;
    // 查询LP token的余额
    uint256 bal = lpToken.balanceOf(address(this));
    // LP token 批准迁移合约控制余额数量
    lpToken.safeApprove(address(migrator), bal);
    // 新LP token地址 = 执行迁移合约的迁移方法
    IERC20 newLpToken = migrator.migrate(lpToken);
    // 确认余额 = 新LP token中的余额
    require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
    // 修改池子信息中的LP token地址为新LP token地址
    pool.lpToken = newLpToken;
}
```

图 16 migrate 函数源码截图

- **相关函数：**migrate、safeApprove、balanceOf
- **安全建议：建议删除该函数**
- **修复结果：已忽略（项目方申明：migrator这个变量的值只能由管理员设置,设置这个的目的是类似于sushiswap的目的,将其他交易所的lptoken对应资产取出,然后存入我们自己的交易所,之后生成我们自己的lptoken这样的目的,要完成这一点首先要设置migrator这个变量,对应的合约也只能是我们自己设计的,因为这个方法权限比较大,所以我们可以将设置migrator的权限交给时间锁合约或者多签合约）**
- **审计结果：通过**

(6) 更新所有抵押池数据功能

- **业务描述：**如下图所示，合约实现了massUpdatePools函数通过遍历所有抵押池，调用updatePool函数来更新所有抵押池信息。

```
 */
// Update reward vairables for all pools. Be careful of gas spending!
function massUpdatePools() public {
    // 池子数量
    uint256 length = poolInfo.length;
    // 遍历所有池子
    for (uint256 pid = 0; pid < length; ++pid) {
        // 升级池子(池子id)
        updatePool(pid);
    }
}
```

图 17 massUpdatePools 函数源码截图

- **相关函数：**updatePool
- **安全建议：无**
- **审计结果：通过**

(7) 更新指定抵押池数据功能

- **业务描述：**如下图所示，合约实现了updatePool函数用于更新指定抵押池信息。调用此函数要求当前区块号大于上一次奖励区块号，并且合约里指定抵押池的LP代币余额大于0。计算上次区块到当前区块获得的GOT奖励，将奖励铸币到本合约地址(要求本合约地址为GOT代币合约owner)，其中GOT合约会额外铸币5%到本合约地址，同时授权给fund地址对应的授权值，在调用fund合约的deposit函数存入开发者准备金，最后更新抵押池的相关参数。

```
function updatePool(uint256 _pid) public {
    // 实例化池子信息
    PoolInfo storage pool = poolInfo[_pid];
    // 如果当前区块号 <= 池子信息.分配发生的最后一个块号
    if (block.number <= pool.lastRewardBlock) {
        // 直接返回
        return;
    }
    // LPtoken的供应量 = 当前合约在`池子信息.lotoken地址`的余额
    uint256 lpSupply = pool.lpToken.balanceOf(address(this));
    // 如果 LPtoken的供应量 == 0
    if (lpSupply == 0) {
        // 池子信息.分配发生的最后一个块号 = 当前块号
        pool.lastRewardBlock = block.number;
        // 返回
        return;
    }
    // 奖金乘积 = 获取奖金乘积(分配发生的最后一个块号，当前块号)
    uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
    // GOT奖励 = 奖金乘积 * 每块创建的GOT令牌 * 池子分配点数 / 总分配点数
    uint256 GOTReward =
        multiplier.mul(GOTPerBlock).mul(pool.allocPoint).div(
            totalAllocPoint
        );
    // 开发者奖励为0.5%
    uint256 fundReserve = GOTReward.div(fundDivisor);
    // 调用GOT的铸造方法，为管理团队铸造 (`GOT奖励` / 20) token
    IGOT(GOT).mint(address(this), fundReserve);
    // 当前合约批准fund地址,开发者准备金数额
    IERC20(GOT).safeApprove(fund, fundReserve);
    // 调用fund合约的存款方法存入开发者准备金
    ISimpleERCFund(fund).deposit(
        GOT,
        fundReserve,
        "MasterChef: Fund Reserve"
    );
    // 调用GOT的铸造方法，为当前合约铸造 `GOT奖励` token
    IGOT(GOT).mint(address(this), GOTReward);
    // 每股累积GOT = 每股累积GOT + GOT奖励 * 1e12 / LPtoken的供应量
    pool.accGOTPerShare = pool.accGOTPerShare.add(
        GOTReward.mul(1e12).div(lpSupply)
    );
    // 池子信息.分配发生的最后一个块号 = 当前块号
    pool.lastRewardBlock = block.number;
}
```

图 18 updatePool 函数源码截图

- **相关函数：**updatePool、balanceOf、getMultiplier、mint、safeApprove
- **安全建议：无**
- **审计结果：通过**

(8) 质押LP代币功能

- **业务描述：**如下图所示，合约实现了deposit函数用于用户质押LP代币获取奖励。用户通过此函数向指定抵押池抵押代币，并更新抵押池信息，如果用户之前在当前抵押池有过抵押，则会计算并领取奖励。抵押代币后更新用户相关参数。

```
function deposit(uint256 _pid, uint256 _amount) public {
    // 实例化池子信息
    PoolInfo storage pool = poolInfo[_pid];
    // 根据池子id和当前用户地址,实例化用户信息
    UserInfo storage user = userInfo[_pid][msg.sender];
    // 将给定池的奖励变量更新为最新
    updatePool(_pid);
    // 如果用户已添加的数额>0
    if (user.amount > 0) {
        // 待定数额 = 用户.已添加的数额 * 池子.每股累积GOT / 1e12 - 用户.已奖励数额
        uint256 pending =
            user.amount.mul(pool.accGOTPerShare).div(1e12).sub(
                user.rewardDebt
            );
        if (pending > 0) {
            // 向当前用户安全发送待定数额的GOT
            safeGOTTransfer(msg.sender, pending);
        }
    }
    if (_amount > 0) {
        // 调用池子.lptoken的安全发送方法,将_amount数额的lp token从当前用户发送到当前合约
        pool.lpToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        // 用户.已添加的数额  = 用户.已添加的数额 + _amount数额
        user.amount = user.amount.add(_amount);
    }
    // 用户.已奖励数额 = 用户.已添加的数额 * 池子.每股累积GOT / 1e12
    user.rewardDebt = user.amount.mul(pool.accGOTPerShare).div(1e12);
    // 触发存款事件
    emit Deposit(msg.sender, _pid, _amount);
}
```

图 19 deposit 函数源码截图

```
// Safe GOT transfer function, just in case if rounding error causes pool
function safeGOTTransfer(address _to, uint256 _amount) internal {
    // GOT余额 = 当前合约在GOT的余额
    uint256 GOTBal = IERC20(GOT).balanceOf(address(this));
    // 如果数额 > GOT余额
    if (_amount > GOTBal) {
        // 按照GOT余额发送GOT到to地址
        IERC20(GOT).safeTransfer(_to, GOTBal);
    } else {
        // 按照_amount数额发送GOT到to地址
        IERC20(GOT).safeTransfer(_to, _amount);
    }
}
```

图 20 safeGOTTransfer 函数源码截图

- **相关函数：**updatePool、balanceOf、getMultiplier、mint、safeApprove、safeGOTTransfer、safeTransferFrom
- **安全建议：无**
- **审计结果：通过**

(9) 提取部分抵押资产功能

- **业务描述：**如下图所示，合约实现了withdraw函数用于实现用户提取指定抵押池已抵押的LP代币。用户调用此函数时，会先更新当前抵押池信息，计算并领取应获得的奖励，提取代币后会更新用户相关参数。

```
function withdraw(uint256 _pid, uint256 _amount) public {
    // 实例化池子信息
    PoolInfo storage pool = poolInfo[_pid];
    // 根据池子id和当前用户地址,实例化用户信息
    UserInfo storage user = userInfo[_pid][msg.sender];
    // 确认用户.已添加数额 >= _amount数额
    require(user.amount >= _amount, "withdraw: not good");
    // 将给定池的奖励变量更新为最新
    updatePool(_pid);
    // 待定数额 = 用户.已添加的数额 * 池子.每股累积GOT / 1e12 - 用户.已奖励数额
    uint256 pending = user.amount.mul(pool.accGOTPerShare).div(1e12).sub(
        user.rewardDebt
    );
    if (pending > 0) {
        // 向当前用户安全发送待定数额的GOT
        safeGOTTransfer(msg.sender, pending);
    }
    if (_amount > 0) {
        // 用户.已添加的数额  = 用户.已添加的数额 - _amount数额
        user.amount = user.amount.sub(_amount);
        // 调用池子.lptoken的安全发送方法,将_amount数额的lp token从当前合约发送到当前用户
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
    }
    // 用户.已奖励数额 = 用户.已添加的数额 * 池子.每股累积GOT / 1e12
    user.rewardDebt = user.amount.mul(pool.accGOTPerShare).div(1e12);
    // 触发提款事件
    emit Withdraw(msg.sender, _pid, _amount);
}
```

图 21 withdraw 函数源码截图

- **相关函数：**updatePool、balanceOf、getMultiplier、mint、safeApprove、safeGOTTransfer、safeTransfer
- **安全建议：无**
- **审计结果：通过**

(10) 紧急提取退出功能

- **业务描述：**如下图所示，合约实现了emergencyWithdraw函数用于紧急提取退出。用户调用此函数会提取出指定抵押池里抵押的所有LP代币，没有奖励计算。

```
function emergencyWithdraw(uint256 _pid) public {
    // 实例化池子信息
    PoolInfo storage pool = poolInfo[_pid];
    // 根据池子id和当前用户地址,实例化用户信息
    UserInfo storage user = userInfo[_pid][msg.sender];
    uint256 amount = user.amount;
    // 用户.已添加数额 = 0
    user.amount = 0;
    // 用户.已奖励数额 = 0
    user.rewardDebt = 0;
    // 调用池子.lptoken的安全发送方法,将_amount数额的lp token从当前合约发送到当前用户
    pool.lpToken.safeTransfer(address(msg.sender), amount);
    // 触发紧急提款事件
    emit EmergencyWithdraw(msg.sender, _pid, amount);
}
```

图 22 emergencyWithdraw 函数源码截图

- **安全建议：无**
- **审计结果：通过**

(11) 领取指定抵押池奖励功能

- **业务描述：**如下图所示，合约实现了harvest函数用于领取指定抵押池奖励。用户调用此函数会领取指定抵押池中的奖励代币。领取奖励后会更新用户相关参数。

```
// Withdraw GOT tokens from MasterChef.
function harvest(uint256 _pid) public {
    // 实例化池子信息
    PoolInfo storage pool = poolInfo[_pid];
    // 根据池子id和当前用户地址,实例化用户信息
    UserInfo storage user = userInfo[_pid][msg.sender];
    // 将给定池的奖励变量更新为最新
    updatePool(_pid);
    // 待定数额 = 用户.已添加的数额 * 池子.每股累积GOT / 1e12 - 用户.已奖励数额
    uint256 pending =
        user.amount.mul(pool.accGOTPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
        // 向当前用户安全发送待定数额的GOT
        safeGOTTransfer(msg.sender, pending);
    }
    // 用户.已奖励数额 = 用户.已添加的数额 * 池子.每股累积GOT / 1e12
    user.rewardDebt = user.amount.mul(pool.accGOTPerShare).div(1e12);
}
```

图 23 harvest 函数源码截图

- **安全建议：无**
- **审计结果：通过**

(12) 提取全部抵押资产功能

- **业务描述：**如下图所示，合约实现了exit函数用于实现用户提取指定抵押池中已抵押所有的LP代币。用户调用此函数时，会先更新当前抵押池信息，计算并领取应获得的奖励，提取代币后会更新用户相关参数。

```
function exit(uint256 _pid) public {
    // 实例化池子信息
    PoolInfo storage pool = poolInfo[_pid];
    // 根据池子id和当前用户地址,实例化用户信息
    UserInfo storage user = userInfo[_pid][msg.sender];
    // 确认用户.已添加数额 >0
    require(user.amount > 0, "withdraw: not good");
    // 将给定池的奖励变量更新为最新
    updatePool(_pid);
    // 待定数额 = 用户.已添加的数额 * 池子.每股累积GOT / 1e12 - 用户.已奖励数额
    uint256 pending =
        user.amount.mul(pool.accGOTPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
        // 向当前用户安全发送待定数额的GOT
        safeGOTTransfer(msg.sender, pending);
    }
    uint256 amount = user.amount;
    // 调用池子.lptoken的安全发送方法,将_amount数额的lp token从当前合约发送到当前用户
    pool.lpToken.safeTransfer(address(msg.sender), amount);
    // 用户.已添加的数额  = 用户.已添加的数额 - _amount数额
    user.amount = 0;
    // 用户.已奖励数额 = 用户.已添加的数额 * 池子.每股累积GOT / 1e12
    user.rewardDebt = amount.mul(pool.accGOTPerShare).div(1e12);
    // 触发提款事件
    emit Withdraw(msg.sender, _pid, amount);
}
```

图 24 exit 函数源码截图

- **相关函数：**updatePool、balanceOf、getMultiplier、mint、safeApprove、safeGOTTransfer、safeTransfer
- **安全建议：无**
- **审计结果：通过**

(13) 查询用户未领取奖励功能

- **业务描述：**如下图所示，合约实现了pendingGOT函数用于实现用户查询指定抵押池中未领取的GOT代币。

```
// View function to see pending GOTs on frontend.
function pendingGOT(uint256 _pid, address _user)
    external
    view
    returns (uint256)
{
    // 实例化池子信息
    PoolInfo storage pool = poolInfo[_pid];
    // 根据池子id和用户地址,实例化用户信息
    UserInfo storage user = userInfo[_pid][_user];
    // 每股累积GOT
    uint256 accGOTPerShare = pool.accGOTPerShare;
    // LPtoken的供应量 = 当前合约在`池子信息.lpToken地址`的余额
    uint256 lpSupply = pool.lpToken.balanceOf(address(this));
    // 如果当前区块号 > 池子信息.分配发生的最后一个块号 && LPtoken的供应量 != 0
    if (block.number > pool.lastRewardBlock && lpSupply != 0) {
        // 奖金乘积 = 获取奖金乘积(分配发生的最后一个块号，当前块号)
        uint256 multiplier =
            getMultiplier(pool.lastRewardBlock, block.number);
        // GOT奖励 = 奖金乘积 * 每块创建的GOT令牌 * 池子分配点数 / 总分配点数
        uint256 GOTReward = multiplier
            .mul(GOTPerBlock)
            .mul(pool.allocPoint)
            .div(totalAllocPoint);
        // 每股累积GOT = 每股累积GOT + GOT奖励 * 1e12 / LPtoken的供应量
        accGOTPerShare = accGOTPerShare.add(
            GOTReward.mul(1e12).div(lpSupply)
        );
    }
    // 返回 用户.已添加的数额 * 每股累积GOT / 1e12 - 用户.已奖励数额
    return user.amount.mul(accGOTPerShare).div(1e12).sub(user.rewardDebt);
}
```

图 25 pendingGOT 函数源码截图

- **相关函数：**balanceOf、getMultiplier
- **安全建议：无**
- **审计结果：通过**

(14) 设置开发者地址功能

- **业务描述：**如下图所示，合约实现了setFund函数用于修改fund地址，仅owner可以调用。

```
     */
    function setFund(address _fund) public onlyOwner {
        fund = _fund;
    }
```

图 26 setFund 函数源码截图

- **安全建议：无**
- **审计结果：通过**

(15) 设置开发者奖励基金比例功能

- **业务描述：**如下图所示，合约实现了setFundDivisor函数用于修改开发者奖励基金比例（默认值为20），仅owner可以调用。

```
function setFundDivisor(uint256 _fundDivisor) public onlyOwner {
    fundDivisor = _fundDivisor;
}
```

图 27 setFundDivisor 函数源码截图

- **安全建议：无**
- **审计结果：通过**

(16) 其他相关查询函数

- **业务描述：**如下图所示，合约实现了poolLength函数用于查询抵押池的数量，getMultiplier函数用于查询奖励计算时的参数，这里设置有12个周期，奖励会随着区块的增加而减少。

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

图 28 poolLength 函数源码截图

```
function getMultiplier(uint256 _from, uint256 _to)
    public
    view
    returns (uint256 multiplier)
{
    // 奖励结束块号
    uint256 bonusEndBlock = startBlock.add(epochPeriod.mul(rewardEpoch));

    // 如果 from块号 >= 奖励结束块号
    if (_from >= bonusEndBlock) {
        // 返回to块号 - from块号
        multiplier = _to.sub(_from);
    // 否则
    } else {
        // from所在的周期 = from距离开始时间过了多少个区块 / 周期区块数量 (取整)
        uint256 fromEpoch = _from.sub(startBlock).div(epochPeriod);
        // to之前的周期 = to距离开始时间过了多少个区块 / 周期区块数量 (取整)
        uint256 toEpoch = _to.sub(startBlock).div(epochPeriod);
        // 如果 to之前的周期 > from所在的周期 说明from和to不在同一个周期内
        if(toEpoch > fromEpoch){
            // from所在的周期内还剩多少个区块 = 周期区块数量 - from距离开始时间过了多少个区块 % 周期区块数量
            uint256 fromEpochBlock = epochPeriod.sub(_from.sub(startBlock).mod(epochPeriod));
            // to剩余的区块 = (to - 开始的区块号) % 周期区块数量
            uint256 toEpochBlock = _to.sub(startBlock).mod(epochPeriod);
            // 乘数 = from所在的周期内还剩多少个区块 * 2 ** (奖励发放的周期数量 - from所在的周期)
            multiplier = fromEpochBlock.mul(2**rewardEpoch.sub(fromEpoch));
            // 从to所在的周期向from所在的周期递减循环
            for (uint256 i = toEpoch; i > fromEpoch; i--) {
                // 幂 = 如果 i >= 奖励发放的周期数量 ? 0 : 奖励发放的周期数量 - i
                uint256 pow = i > rewardEpoch ? 0 : rewardEpoch.sub(i);
                // 乘数 = 乘数 + 每个周期的区块数量 * 2 ** 幂
                multiplier = multiplier.add(epochPeriod.mul(2**pow));
            }
            // 如果 to之前的周期 < 奖励结束块号
            if (toEpoch < rewardEpoch) {
                // 乘数 = 乘数 + to剩余的区块 * 2 ** (奖励发放的周期数量 - to之前的周期 )
                multiplier = multiplier.add(
                    toEpochBlock.mul(2**rewardEpoch.sub(toEpoch))
                );
            } else {
                // 乘数 = 乘数 + to剩余的区块
                multiplier = multiplier.add(toEpochBlock);
            }
        // 否则from和to在同一个周期内
        }else{
            // 乘数 = (to - from) * 2 ** (奖励发放的周期数量 - to之前的周期)
            multiplier = _to.sub(_from).mul(2**(rewardEpoch.sub(toEpoch)));
        }
    }
```

图 29 getMultiplier 函数源码截图

- **安全建议：无**
- **审计结果：通过**

1. 结论

Beosin(成都链安)对 Goswap 项目的智能合约的设计和代码实现进行了详细的审计。审计团队在审计过程中发现的问题均已告知项目方并就修复结果达成一致，Goswap 项目的智能合约的总体审计结果是**通过。**

成都链安
BEOSIN

官方网址

https://lianantech.com

电子邮箱

vaas@lianantech.com

微信公众号