降维安全
JOHNWICK SECURITY

守 护 价 值 互 联 网

TRON
审 计 报 告

降 维 安 全 实 验 室

WWW.JOHNWICK.IO

降维安全实验室于 **2020** 年 **9** 月 **27** 日 收到 **PeachPool**（公司/团队）**PCH** 项目智能合约源代码安全审计需求。

项目名称：PeachPool(PCH)

合约地址：

https://tronscan.org/#/contract/TCH2eTw5GFMo3gB8hSdsKTdiUwwcePDTNi/code

审计编号：20200915

审计日期：20200928

审计项目及结果：

审计大类	审计子类	审计结果（通过/未通过）
合约漏洞	整数溢出	通过
	竞争条件	通过
	拒绝服务	通过
	逻辑漏洞	通过
	硬编码地址	通过
	函数参数检查	未通过
	函数访问权限绕过	通过
	随机数生成	未通过
	随机数使用	通过
合约规范	编译器版本指定	通过
	事件记录	通过
	回退函数使用	通过
	构造函数使用	通过
	函数可见性声明	通过
	变量存储位置声明	通过
	废弃关键字使用	通过
	TRC20/223 标准	通过
	TRC721 标准	通过
业务风险	任意增发	通过
	任意销毁	通过
	任意暂停转账	通过
	"短地址"攻击	通过
	"假充值"攻击	通过
GAS 优化	assert()/require()	通过
	for/while 循环优化	通过
	storage 优化	通过
自动化模糊测试		通过

(其他未知安全漏洞和 TRON 波场公链设计缺陷不包含在本次审计责任范围内)

审计结果：通过

审计团队：降维安全实验室

（声明：降维安全实验室依据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。对于本报告出具后，发生或存在的事实，降维安全实验室无法判断其智能合约安全状况，亦不对此承担任何责任。本报告所做的安全审计分析及其他内容，基于信息提供者截止本报告出具时向降维安全实验室提供的相关材料和文件（简称已提供资料）。降维安全实验室假设：已提供资料不存在缺失、篡改、删减、隐瞒的情况。如已提供资料存在信息缺失、被篡改、被删减、被隐瞒的情况，或资料提供者反应的情况与实际情况不符的，降维安全实验室对由此导致的损失和不利影响不承担任何责任。）

审计详情：

//JohnWick:
本合约创建了符合 TRC20 标准的 mineToken 代币 PeachPool(PCH),并通过 MineFactory 合约类的 `createMine` 函数动态生成不同的质押合约来质押不同 stakeToken 代币,挖取 mineToken 代币.

//JohnWick: 114L 281L 945L
本合约使用了 `SafeMath, SignedSafeMath` 函数库来避免潜在的整数溢出问题,符合推荐的做法.

//JohnWick: **[Low Risk]** 734L
`stake(uint256 amount, address referrer)` 调用
`doStake(amount, referrer)`
在 doStake 函数中并未检查 `referrer` 是否为 `msg.sender`,也就是可以自己推广自己拿返佣,建议加上检查

```
if (referrer != address(0) &&
    referrer  != msg.sender &&
    referrers[msg.sender] == address(0) && )  {
    ...
}
```

//JohnWick: **[Low Risk]**
`getRandomSeed()` 用于生成计算 Jackpot 所采用的随机数,随机源有两个 `blockhash(block.number - 1)` 和 `currentBlockCallSequence` :
- `blockhash(block.number - 1)` 是上个区块的 hash
- `currentBlockCallSequence` 由函数修饰符 `updateRandomnessMeta` 负责更新:
每个区块第一次调用 `updateRandomnessMeta`
将 `currentBlockCallSequence` 重置为 0,
同一区块每再调用一次,`currentBlockCallSequence` 值加 1.

如果**波场矿工调整交易打包顺序**
可以控制 `currentBlockCallSequence` 的取值,从而预测随机数和 `roll` 的值,并最终将奖池中 `jackpotTrxAmount` 个 TRX 转到攻击者控制的 `staker` 账号里.

注: 审计详情中所涉及的代码行号均基于项目方上传于 `tronscan.org` 的已验证合约源代码,即后附的`审计源码`.

审计源码:

```
pragma solidity ^0.5.0;

library Math {
    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a < b ? a : b;
    }
}

library SafeMath {
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    function sub(
        uint256 a,
        uint256 b,
        string memory errorMessage
    ) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }
```

```
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    function div(
        uint256 a,
        uint256 b,
        string memory errorMessage
    ) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;

        return c;
    }
}

library SignedSafeMath {
    int256 private constant _INT256_MIN = -2**255;

    function mul(int256 a, int256 b) internal pure returns (int256) {
        if (a == 0) {
            return 0;
        }

        require(
            !(a == -1 && b == _INT256_MIN),
            "SignedSafeMath: multiplication overflow"
        );

        int256 c = a * b;
        require(c / a == b, "SignedSafeMath: multiplication overflow");

        return c;
```

```
    }
}

interface IERC20 {
    function totalSupply() external view returns (uint256);

    function balanceOf(address account) external view returns (uint256);

    function transfer(address recipient, uint256 amount)
        external
        returns (bool);

    function allowance(address owner, address spender)
        external
        view
        returns (uint256);

    function approve(address spender, uint256 amount) external returns (bool);

    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool);

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

contract ERC20 is IERC20 {
    using SafeMath for uint256;

    mapping(address => uint256) private _balances;

    mapping(address => mapping(address => uint256)) private _allowances;

    uint256 private _totalSupply;

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

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

    function transfer(address recipient, uint256 amount) public returns (bool)
{
        _transfer(msg.sender, recipient, amount);
        return true;
    }

    function allowance(address owner, address spender)
        public
        view
        returns (uint256)
    {
        return _allowances[owner][spender];
    }

    function approve(address spender, uint256 amount) public returns (bool) {
        _approve(msg.sender, spender, amount);
        return true;
    }

    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) public returns (bool) {
        _transfer(sender, recipient, amount);
        _approve(
            sender,
            msg.sender,
            _allowances[sender][msg.sender].sub(
                amount,
                "ERC20: transfer amount exceeds allowance"
            )
        );
        return true;
    }

    function increaseAllowance(address spender, uint256 addedValue)
        public
        returns (bool)
    {
```

```
        _approve(
            msg.sender,
            spender,
            _allowances[msg.sender][spender].add(addedValue)
        );
        return true;
    }

    function decreaseAllowance(address spender, uint256 subtractedValue)
        public
        returns (bool)
    {
        _approve(
            msg.sender,
            spender,
            _allowances[msg.sender][spender].sub(
                subtractedValue,
                "ERC20: decreased allowance below zero"
            )
        );
        return true;
    }

    function _transfer(
        address sender,
        address recipient,
        uint256 amount
    ) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _balances[sender] = _balances[sender].sub(
            amount,
            "ERC20: transfer amount exceeds balance"
        );
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }

    function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address");

        _totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(address(0), account, amount);
```

```
    }

    function _burn(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: burn from the zero address");

        _balances[account] = _balances[account].sub(
            amount,
            "ERC20: burn amount exceeds balance"
        );
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), amount);
    }

    function _approve(
        address owner,
        address spender,
        uint256 amount
    ) internal {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    function _burnFrom(address account, uint256 amount) internal {
        _burn(account, amount);
        _approve(
            account,
            msg.sender,
            _allowances[account][msg.sender].sub(
                amount,
                "ERC20: burn amount exceeds allowance"
            )
        );
    }
}

contract ERC20Detailed is IERC20 {
    string private _name;
    string private _symbol;
    uint8 private _decimals;

    constructor(
        string memory name,
```

```
        string memory symbol,
        uint8 decimals
    ) public {
        _name = name;
        _symbol = symbol;
        _decimals = decimals;
    }

    function name() public view returns (string memory) {
        return _name;
    }

    function symbol() public view returns (string memory) {
        return _symbol;
    }

    function decimals() public view returns (uint8) {
        return _decimals;
    }
}

contract Mine {
    using SafeMath for uint256;
    using SignedSafeMath for int256;

    event Staked(address indexed staker, uint256 indexed amount);
    event Unstaked(address indexed staker, uint256 indexed amount);
    event Referral(address indexed referrer, address indexed referee);
    event RewardClaimed(address indexed staker, uint256 indexed amount);
    event ReferralReward(
        address indexed referrer,
        address indexed referee,
        uint256 indexed amount
    );
    event EnergyChanged(
        address indexed staker,
        uint256 indexed amountBefore,
        uint256 indexed amountAfter,
        uint256 rawReward,
        uint256 baseEnergy,
        uint8 effect
    );
    event JackpotTaken(
        address indexed winner,
        uint256 indexed trxAmount,
```

```
    uint256 indexed tokenAmount,
    bool isRecycle
);
event Abandoned();

address public factory;
address payable public developer;
address public mineToken;
address public stakeToken;
bool public initialized;

/* Pool Settings */
uint256 public startTime;
uint256 public endTime;
uint256 public poolRewardAmount;
uint256 public referralRewardRatio;
uint256 public devRewardRatio;
uint256 public jackpotEnergyThreshold;
uint256 public energyBaseLimit;
uint256 public jackpotTrxAmount;
uint256 public jackpotMineTokenAmount;

/* Live Pool Data */
bool public isAbandoned;
bool public isJackpotTaken;
uint256 public currentEnergy;
uint256 public totalStakeAmount;
mapping(address => uint256) public stakeAmounts;
mapping(address => uint256) public rewards;
mapping(address => address) public referrers;
uint256 public accuDevReward;

/* Reward Tracking Data */
uint256 private outputPerSecond;
uint256 private lastTokenRewardRate;
uint256 private lastTokenRewardRateUpdateTime;
mapping(address => uint256) private stakerEntryRewardRates;

/* Randomness Metadata */
uint256 private lastCallBlockNumber;
uint256 private currentBlockCallSequence;

uint256 public constant RATIO_ONE = 10**4; // 100.00%
uint256 public constant MAX_DEV_REWARD = 10**3; // 10.00%
uint256 public constant JACKPOT_RECYCLE_DELAY = 1 days;
```

```
    uint256 public constant ENERGY_EFFECT_1_CHANCE = 500; // 5.00% -> Negate
2x
    uint256 public constant ENERGY_EFFECT_2_CHANCE = 1000; // 10.00% -> Negate
1x
    uint256 public constant ENERGY_EFFECT_3_CHANCE = 2000; // 20.00% -> 2x
    uint256 public constant ENERGY_EFFECT_4_CHANCE = 1000; // 10.00% -> 3x
    uint256 public constant ENERGY_EFFECT_5_CHANCE = 200; // 2.00% -> 10x
    uint256 public constant TOKEN_REWARD_RATE_MULTIPLIER = 10**30; // To avoi
precision loss

    modifier onlyFactory() {
        require(msg.sender == factory, "Mine: not factory");
        _;
    }

    modifier onlyDeveloper() {
        require(msg.sender == developer, "Mine: not developer");
        _;
    }

    modifier onlyInitialized() {
        require(initialized, "Mine: not initialized");
        _;
    }

    modifier onlyNotAbandoned() {
        require(!isAbandoned, "Mine: abandoned");
        _;
    }

    modifier onlyStarted() {
        require(block.timestamp >= startTime, "Mine: not started");
        _;
    }

    modifier onlyNotStarted() {
        require(block.timestamp < startTime, "Mine: started");
        _;
    }

    modifier onlyNotEnded() {
        require(block.timestamp < endTime, "Mine: already ended");
        _;
    }
```

```
    modifier onlyEnded() {
        require(block.timestamp >= endTime, "Mine: not ended");
        _;
    }

    modifier updateRandomnessMeta() {
        if (lastCallBlockNumber == block.number) {
            // Not the first call (no SafeMath needed to save tx cost)
            currentBlockCallSequence = currentBlockCallSequence + 1;
        } else {
            // The first call in this block
            lastCallBlockNumber = block.number;
            currentBlockCallSequence = 0;
        }
        _;
    }

    modifier updateTokenRewardRate() {
        uint256 appliedUpdateTime = Math.min(block.timestamp, endTime);
        uint256 durationInSeconds = appliedUpdateTime.sub(
            lastTokenRewardRateUpdateTime
        );

        // This saves tx cost when being called multiple times in the same block
        if (durationInSeconds > 0) {
            // No need to update the rate if no one staked at all
            if (totalStakeAmount > 0) {
                lastTokenRewardRate = lastTokenRewardRate.add(

durationInSeconds.mul(outputPerSecond).div(totalStakeAmount)
                );
            }
            lastTokenRewardRateUpdateTime = appliedUpdateTime;
        }
        _;
    }

    /**
     * @dev updateTokenRewardRate() is always called before this modifier.
     * So lastTokenRewardRate and lastTokenRewardRateUpdateTime are accurate.
     */
    modifier updateStakerReward(address staker) {
        uint256 stakeAmount = stakeAmounts[staker];
        uint256 stakerEntryRate = stakerEntryRewardRates[staker];
        uint256 rateDifference = lastTokenRewardRate.sub(stakerEntryRate);
```

```
        if (rateDifference > 0) {
            rewards[staker] = rewards[staker].add(
                stakeAmount.mul(rateDifference).div(
                    TOKEN_REWARD_RATE_MULTIPLIER
                )
            );
            stakerEntryRewardRates[staker] = lastTokenRewardRate;
        }
        _;
    }

    /**
     * @dev Basically just updateTokenRewardRate() + updateStakerReward(), but
     * in a non-cost-saving way. This function is only used by external
applications
     */
    function getReward(address staker) external view returns (uint256) {
        uint256 latestTokenRewardRate = totalStakeAmount > 0
            ? lastTokenRewardRate.add(
                Math
                    .min(block.timestamp, endTime)
                    .sub(lastTokenRewardRateUpdateTime)
                    .mul(outputPerSecond)
                    .div(totalStakeAmount)
            )
            : lastTokenRewardRate;

        return
            rewards[staker].add(
                stakeAmounts[staker]
                    .mul(
                    latestTokenRewardRate.sub(stakerEntryRewardRates[staker])
                )
                    .div(TOKEN_REWARD_RATE_MULTIPLIER)
            );
    }

    constructor(
        address payable _developer,
        address _mineToken,
        address _stakeToken
    ) public {
        require(
            _developer != address(0) &&
```

```
            _mineToken != address(0) &&
            _stakeToken != address(0),
        "Mine: zero address"
    );
    require(_mineToken != _stakeToken, "Mine: same token");

    factory = msg.sender;
    developer = _developer;
    mineToken = _mineToken;
    stakeToken = _stakeToken;
    initialized = false;
}

function() external payable {
    require(msg.sender == factory, "Mine: only factory can pay");
    require(msg.data.length == 0, "Mine: invalid fallback");
}

function initialize(
    uint256 _startTime,
    uint256 _endTime,
    uint256 _poolRewardAmount,
    uint256 _referralRewardRatio,
    uint256 _devRewardRatio,
    uint256 _jackpotEnergyThreshold,
    uint256 _energyBaseLimit,
    uint256 _jackpotTrxAmount,
    uint256 _jackpotMineTokenAmount
) external onlyFactory updateRandomnessMeta {
    require(!initialized, "Mine: already initialized");
    require(
        _startTime > block.timestamp && _endTime > _startTime,
        "Mine: invalid time range"
    );
    require(_poolRewardAmount > 0, "Mine: pool reward cannot be zero");
    require(
        _referralRewardRatio <= _devRewardRatio,
        "Mine: invalid referral reward ratio"
    );
    require(
        _devRewardRatio <= MAX_DEV_REWARD,
        "Mine: invalid dev reward ratio"
    );
    require(
        address(this).balance >= _jackpotTrxAmount,
```

```
            "Mine: insufficient TRX for jackpot"
        );

        // Jackpot related checks
        if (_jackpotEnergyThreshold > 0) {
            // Setting a jackpot

            require(
                _energyBaseLimit > 0,
                "Mine: energy base limit must be positive unless not setting a
jackpot"
            );
            require(
                _jackpotTrxAmount > 0 || _jackpotMineTokenAmount > 0,
                "Mine: jackpot empty"
            );
        } else {
            // Not setting a jackpot

            require(
                _energyBaseLimit == 0,
                "Mine: energy base limit must be zero when not setting a jackpot"
            );
            require(
                _jackpotTrxAmount == 0 && _jackpotMineTokenAmount == 0,
                "Mine: jackpot amounts must empty when not setting a jackpot"
            );
        }

        initialized = true;

        // Take all mine tokens, including pool, dev reward, and jackpot here
now (instead of minting on the go)
        uint256 totalMineTokenNeeded = _poolRewardAmount
            .mul(RATIO_ONE.add(_devRewardRatio))
            .div(RATIO_ONE)
            .add(_jackpotMineTokenAmount);
        require(
            IERC20(mineToken).transferFrom(
                factory,
                address(this),
                totalMineTokenNeeded
            ),
            "Mine: failed to take mine token from factory"
        );
```

```
        startTime = _startTime;
        endTime = _endTime;
        poolRewardAmount = _poolRewardAmount;
        referralRewardRatio = _referralRewardRatio;
        devRewardRatio = _devRewardRatio;
        jackpotEnergyThreshold = _jackpotEnergyThreshold;
        energyBaseLimit = _energyBaseLimit;
        jackpotTrxAmount = _jackpotTrxAmount;
        jackpotMineTokenAmount = _jackpotMineTokenAmount;

        outputPerSecond = _poolRewardAmount
            .mul(TOKEN_REWARD_RATE_MULTIPLIER) // avoid precision loss
            .div(_endTime.sub(_startTime));
        lastTokenRewardRate = 0;
        lastTokenRewardRateUpdateTime = _startTime;
    }

    function stake(uint256 amount, address referrer)
        external
        onlyInitialized
        onlyNotAbandoned
        onlyStarted
        onlyNotEnded
        updateRandomnessMeta
        updateTokenRewardRate
        updateStakerReward(msg.sender)
    {
        doStake(amount, referrer);
    }

    function unstake(uint256 amount)
        external
        onlyInitialized
        onlyStarted
        updateRandomnessMeta
        updateTokenRewardRate
        updateStakerReward(msg.sender)
    {
        doUnstake(amount);
    }

    function unstakeAll()
        external
        onlyInitialized
```

```
        onlyStarted
        updateRandomnessMeta
        updateTokenRewardRate
        updateStakerReward(msg.sender)
    {
        doUnstake(stakeAmounts[msg.sender]);
    }

    function claimReward()
        external
        onlyInitialized
        onlyStarted
        updateRandomnessMeta
        updateTokenRewardRate
        updateStakerReward(msg.sender)
    {
        doClaimReward(msg.sender);
    }

    function claimDevReward(uint256 amount)
        external
        onlyInitialized
        onlyStarted
        onlyDeveloper
        updateRandomnessMeta
    {
        require(amount > 0, "Mine: cannot claim zero reward");

        accuDevReward = accuDevReward.sub(amount);
        require(
            IERC20(mineToken).transfer(developer, amount),
            "Mine: token transfer failed"
        );
    }

    function sendReward(address payable staker)
        external
        onlyInitialized
        onlyEnded
        onlyDeveloper
        updateRandomnessMeta
        updateTokenRewardRate
        updateStakerReward(staker)
    {
        doClaimReward(staker);
```

```
    }

    function recycleJackpot()
        external
        onlyInitialized
        onlyDeveloper
        updateRandomnessMeta
    {
        require(jackpotEnergyThreshold > 0, "Mine: no jackpot");
        require(!isJackpotTaken, "Mine: jackpot already taken");
        require(
            isAbandoned || (block.timestamp > endTime + JACKPOT_RECYCLE_DELAY),
            "Mine: recycle time not reached"
        );

        isJackpotTaken = true;
        if (jackpotTrxAmount > 0) developer.transfer(jackpotTrxAmount);
        if (jackpotMineTokenAmount > 0)
            require(
                IERC20(mineToken).transfer(developer,
jackpotMineTokenAmount),
                "Mine: token transfer failed"
            );

        emit JackpotTaken(
            developer,
            jackpotTrxAmount,
            jackpotMineTokenAmount,
            true
        );
    }

    /**
     * @dev Enables the developer to abandon the pool before it starts so that
the jackpot
     * can be recycled immediately.
     */
    function abandonMine()
        external
        onlyInitialized
        onlyNotStarted
        onlyDeveloper
    {
        require(!isAbandoned, "Mine: already abandoned");
        isAbandoned = true;
```

```
        emit Abandoned();
    }

    /**
     * @dev Take back pool tokens after it's abandoned
     */
    function retrieveMineToken(uint256 amount) external onlyDeveloper {
        require(isAbandoned, "Mine: not abandoned");

        // No need to assert success
        IERC20(mineToken).transfer(developer, amount);
    }

    /**
     * @dev This function is for emergency only. Just in case the contract
deadlocks somehow,
     * we can at least send the staked amounts back.
     */
    function emergencyExit(address staker) external onlyDeveloper {
        uint256 stakeAmount = stakeAmounts[staker];
        require(stakeAmount > 0, "Mine: nothing to return");

        // Cleanup (we won't event touch totalStakeAmount to avoid underflow)
        stakeAmounts[staker] = 0;
        stakerEntryRewardRates[staker] = lastTokenRewardRate;

        // We don't even require this transfer to succeed
        IERC20(stakeToken).transfer(staker, stakeAmount);
    }

    function doStake(uint256 amount, address referrer) private {
        require(amount > 0, "Mine: cannot stake zero amount");

        // Referral relationship can only be set once
        if (referrer != address(0) && referrers[msg.sender] == address(0)) {
            referrers[msg.sender] = referrer;
            emit Referral(referrer, msg.sender);
        }

        stakeAmounts[msg.sender] = stakeAmounts[msg.sender].add(amount);
        totalStakeAmount = totalStakeAmount.add(amount);

        require(
            IERC20(stakeToken).transferFrom(msg.sender,        address(this),
```

```
amount),
            "Mine: token transfer failed"
        );

        emit Staked(msg.sender, amount);
    }

    function doUnstake(uint256 amount) private {
        require(amount > 0, "Mine: cannot unstake zero amount");

        // No sufficiency check required as sub() will throw anyways
        stakeAmounts[msg.sender] = stakeAmounts[msg.sender].sub(amount);
        totalStakeAmount = totalStakeAmount.sub(amount);

        require(
            IERC20(stakeToken).transfer(msg.sender, amount),
            "Mine: token transfer failed"
        );

        emit Unstaked(msg.sender, amount);
    }

    function doClaimReward(address payable staker) private {
        uint256 rewardToClaim = rewards[staker];
        require(rewardToClaim > 0, "Mine: cannot claim zero reward");

        rewards[staker] = 0;

        require(
            IERC20(mineToken).transfer(staker, rewardToClaim),
            "Mine: token transfer failed"
        );

        emit RewardClaimed(staker, rewardToClaim);

        // Calculate referral reward
        address referrer = referrers[staker];
        uint256 referralRewardAmount = referrer == address(0)
            ? 0
            : rewardToClaim.mul(referralRewardRatio).div(RATIO_ONE);

        // Settle referral reward by sending the amount to referrer directly
        if (referralRewardAmount > 0) {
            require(
                IERC20(mineToken).transfer(referrer, referralRewardAmount),
```

```
                "Mine: token transfer failed"
            );

            emit ReferralReward(referrer, staker, referralRewardAmount);
        }

        // Calculate dev reward (referral rewards (if any) are taken from
developer rewards)
        uint256 devRewardAmount = rewardToClaim
            .mul(devRewardRatio)
            .div(RATIO_ONE)
            .sub(referralRewardAmount);

        // Settle developer reward by recording it without sending (save staker
tx cost)
        if (devRewardAmount > 0) {
            accuDevReward = accuDevReward.add(devRewardAmount);
        }

        // Jackpot calculations
        if (jackpotEnergyThreshold > 0 && !isJackpotTaken) {
            uint256 energyBefore = currentEnergy;
            uint256 baseEnergy = Math.min(rewardToClaim, energyBaseLimit);

            // Calculate energy delta
            int256 energyDelta;
            uint8 energyEffectId = 0;
            {
                energyDelta = int256(baseEnergy);

                if (energyDelta < 0) {
                    // rewardToClaim was so large that the highest bit is 1.
                    // Ignore this energy change directly
                    energyDelta = 0;
                } else {
                    // Apply special effect
                    // roll ~ [0, RATIO_ONE - 1]
                    uint256 roll = getRandomSeed() % RATIO_ONE;
                    if (roll < ENERGY_EFFECT_1_CHANCE) {
                        // Negate 2x
                        energyDelta = (-energyDelta).mul(2);
                        energyEffectId = 1;
                    } else if (
                        roll < ENERGY_EFFECT_1_CHANCE + ENERGY_EFFECT_2_CHANCE
                    ) {
```

```
                // Negate 1x
                energyDelta = -energyDelta;
                energyEffectId = 2;
            } else if (
                roll <
                ENERGY_EFFECT_1_CHANCE +
                    ENERGY_EFFECT_2_CHANCE +
                    ENERGY_EFFECT_3_CHANCE
            ) {
                // 2x
                energyDelta = energyDelta.mul(2);
                energyEffectId = 3;
            } else if (
                roll <
                ENERGY_EFFECT_1_CHANCE +
                    ENERGY_EFFECT_2_CHANCE +
                    ENERGY_EFFECT_3_CHANCE +
                    ENERGY_EFFECT_4_CHANCE
            ) {
                // 3x
                energyDelta = energyDelta.mul(3);
                energyEffectId = 4;
            } else if (
                roll <
                ENERGY_EFFECT_1_CHANCE +
                    ENERGY_EFFECT_2_CHANCE +
                    ENERGY_EFFECT_3_CHANCE +
                    ENERGY_EFFECT_4_CHANCE +
                    ENERGY_EFFECT_5_CHANCE
            ) {
                // 10x
                energyDelta = energyDelta.mul(10);
                energyEffectId = 5;
            }
        }
    }

    if (energyDelta >= 0) {
        // Energy increases (conversion is safe)
        currentEnergy = energyBefore.add(uint256(energyDelta));

        if (
            energyBefore < jackpotEnergyThreshold &&
            currentEnergy >= jackpotEnergyThreshold
        ) {
```

```
                // Take the jackpot!
                isJackpotTaken = true;

                // Settle TRX
                if (jackpotTrxAmount > 0) {
                    staker.transfer(jackpotTrxAmount);
                }

                // Settle mine token
                if (jackpotMineTokenAmount > 0) {
                    require(
                        IERC20(mineToken).transfer(
                            staker,
                            jackpotMineTokenAmount
                        ),
                        "Mine: token transfer failed"
                    );
                }

                emit JackpotTaken(
                    staker,
                    jackpotTrxAmount,
                    jackpotMineTokenAmount,
                    false
                );
            }
        } else {
            // Energy decreases
            uint256 energyToSubtract = uint256(-energyDelta);

            currentEnergy = currentEnergy <= energyToSubtract
                ? 0
                : currentEnergy.sub(energyToSubtract);

            // Impossible to take the jackpot. No need to check
        }

        emit EnergyChanged(
            staker,
            energyBefore,
            currentEnergy,
            rewardToClaim,
            baseEnergy,
            energyEffectId
        );
```

```
        }
    }

    /**
     * @dev This is NOT true randomness, but should be good enough for this case.
     * @return uint256
     */
    function getRandomSeed() private view returns (uint256) {
        return
            uint256(
                sha256(
                    abi.encode(
                        blockhash(block.number - 1),
                        currentBlockCallSequence
                    )
                )
            );
    }
}

/**
 * @dev This contract is both a token contract and and a factory contract.
 */
contract MineFactory is ERC20, ERC20Detailed("PeachPool", "PCH", 18) {
    using SafeMath for uint256;

    event MineCreated(address indexed mineAddress);

    address payable public developer;

    uint256 public constant RATIO_ONE = 10**4; // 100.00%

    constructor(uint256 _totalSupply) public {
        require(_totalSupply > 0, "MineFactory: total supply must be positive");

        developer = msg.sender;
        _mint(msg.sender, _totalSupply);
    }

    function createMine(
        address stakeToken,
        uint256 startTime,
        uint256 endTime,
        uint256 poolRewardAmount,
        uint256 referralRewardRatio,
```

```
        uint256 devRewardRatio,
        uint256 jackpotEnergyThreshold,
        uint256 energyBaseLimit,
        uint256 jackpotTrxAmount,
        uint256 jackpotMineTokenAmount
    ) external payable returns (address) {
        require(msg.sender == developer, "MineFactory: not developer");
        require(
            jackpotTrxAmount == msg.value,
            "MineFactory: jackpot TRX amount mismatch"
        );

        // Need to take this amount from caller first
        uint256 totalMineTokenNeeded = poolRewardAmount
            .mul(RATIO_ONE.add(devRewardRatio))
            .div(RATIO_ONE)
            .add(jackpotMineTokenAmount);
        IERC20(address(this)).transferFrom(
            msg.sender,
            address(this),
            totalMineTokenNeeded
        );

        // Create the contract first to get the address
        Mine newMine = new Mine(developer, address(this), stakeToken);

        if (msg.value > 0)
            address(uint160(address(newMine))).transfer(msg.value);
        IERC20(address(this)).approve(address(newMine),
totalMineTokenNeeded);

        newMine.initialize(
            startTime,
            endTime,
            poolRewardAmount,
            referralRewardRatio,
            devRewardRatio,
            jackpotEnergyThreshold,
            energyBaseLimit,
            jackpotTrxAmount,
            jackpotMineTokenAmount
        );

        emit MineCreated(address(newMine));
```

```
        return address(newMine);
    }
}
```