hexens × QUICKSWAP

SMART CONTRACT AUDIT REPORT FOR QUICKSWAP

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

KASPER ZWIJSEN

Lead Smart Contract Auditor | Hexens

Audit Starting Date
28.06.2023

Audit Completion Date
06.07.2023

---

hexens × QUICKSWAP

METHODOLOGY

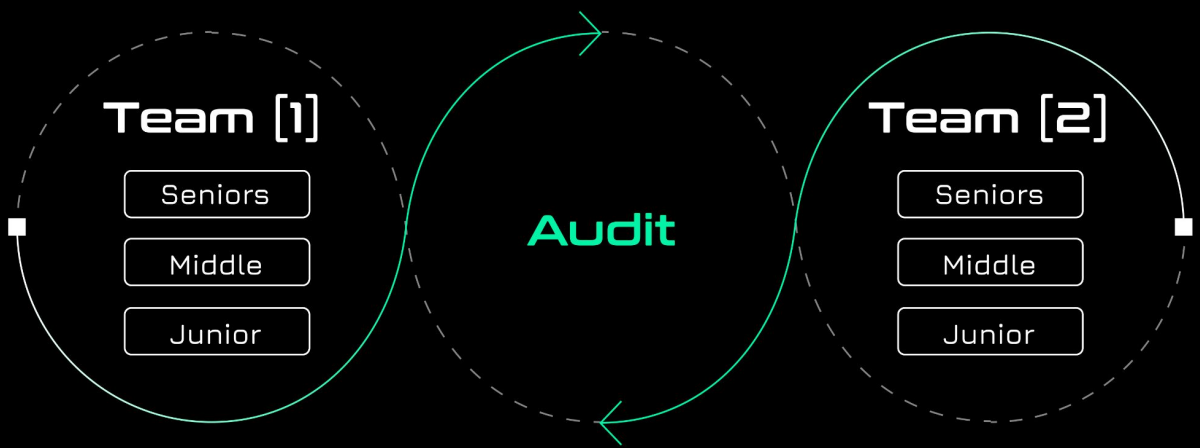
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor* Audit

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered an update to QuickSwap's perpetual futures exchange, QuickPerps. This update added the functionality of multiple reward tokens in the RewardDistributor contract and corresponding changes to other dependent contracts.

Our security assessment was a full review of the new functionality and its effects on existing implementation.

During our audit, we have identified 1 high severity vulnerability in the RewardRouter. It would allow an attacker to block transfers between users.

We have also identified 1 medium severity vulnerability, various minor vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:
https://github.com/QuickSwap/perps/tree/3866d95e1c3190f05e863630b31d9ba9b14fbb29

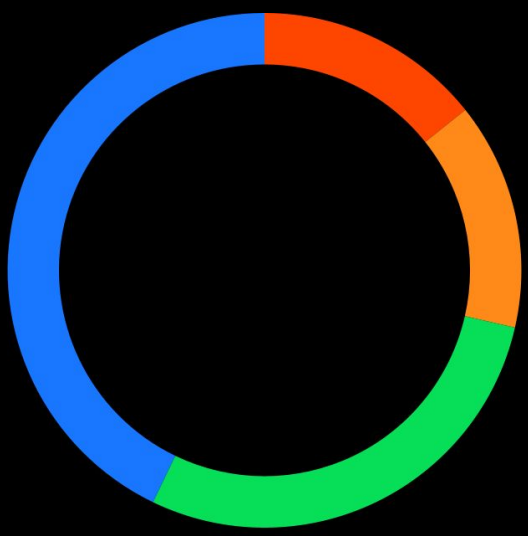
The issues described in this report were fixed in the following commits:
https://github.com/QuickSwap/perps/commit/26b7166d0b2c03884ec062f2329e42b22bf50b4e
https://github.com/QuickSwap/perps/commit/95e8de2accfe1088dd4a37b7f462e260a6ee7210

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	1
MEDIUM	1
LOW	2
INFORMATIONAL	3

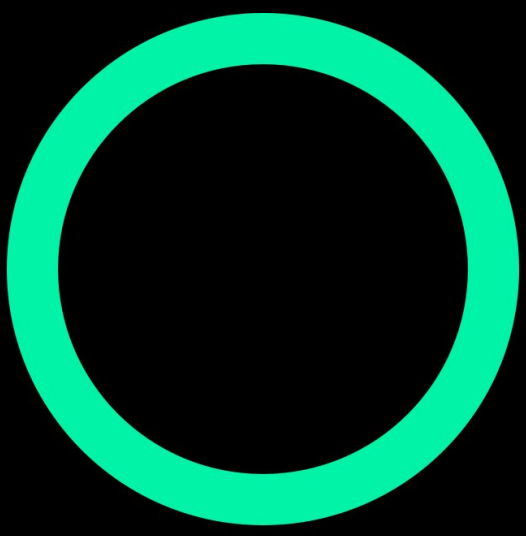
TOTAL: 7

SEVERITY



High Medium Low Informational

STATUS



Fixed

WEAKNESSES

This section contains the list of discovered weaknesses.

QSWP-39. REWARDROUTER TRANSFERS CAN BE BLOCKED

SEVERITY: High

PATH: RewardRouter.sol:acceptTransfer:L237-248

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The RewardRouter implements transfer functionality for stake in the RewardTracker. The sender can send their using **signalTransfer** and consequently the receiver has to accept the transfer with **acceptTransfer**.

The **acceptTransfer** uses the **_validateReceiver** function, which checks that the receiver does not have an average stake or accrued rewards in the RewardTracker, otherwise the transfer will revert.

As such, it becomes possible to back-run the **signalTransfer** call and transfer a small amount using **StakedQlp.transfer** to the intended receiver. If the amount and time is enough, the receiver will receive a tiny amount of rewards (e.g. 1 Wei would be enough) and the legit transfer would get blocked.

RewardRouter.sol

```
    function signalTransfer(address _receiver) external nonReentrant {
        _validateReceiver(_receiver);
        pendingReceivers[msg.sender] = _receiver;
    }

    function acceptTransfer(address _sender) external nonReentrant {
        address receiver = msg.sender;
        require(pendingReceivers[_sender] == receiver, "RewardRouter: transfer not signalled");
        delete pendingReceivers[_sender];

        _validateReceiver(receiver);
        uint256 qlpAmount = IRewardTracker(feeQlpTracker).depositBalances(_sender, qlp);
        if (qlpAmount > 0) {
            IRewardTracker(feeQlpTracker).unstakeForAccount(_sender, qlp, qlpAmount, _sender);
            IRewardTracker(feeQlpTracker).stakeForAccount(_sender, receiver, qlp, qlpAmount);
        }
    }

    function _validateReceiver(address _receiver) private view {
    require(IRewardTracker(feeQlpTracker).averageStakedAmounts(_receiver) == 0, "RewardRouter:
feeQlpTracker.averageStakedAmounts > 0");
        require(!IRewardTracker(feeQlpTracker).hasCumulativeRewards(_receiver), "RewardRouter:
feeQlpTracker.cumulativeRewards > 0");
    }
```

RewardTracker.sol

```
function stakeForAccount(
    address _fundingAccount,
    address _account,
    address _depositToken,
    uint256 _amount
) external override nonReentrant {
    _validateHandler();
    _stake(_fundingAccount, _account, _depositToken, _amount);
}

function _validateHandler() private view {
    require(isHandler[msg.sender], "RewardTracker: forbidden");
}

function _stake(
    address _fundingAccount,
    address _account,
    address _depositToken,
    uint256 _amount
) private {
    require(_amount > 0, "RewardTracker: invalid _amount");
    require(isDepositToken[_depositToken], "RewardTracker: invalid _depositToken");

    IERC20(_depositToken).safeTransferFrom(_fundingAccount, address(this), _amount);

    _updateRewardsAll(_account);

    stakedAmounts[_account] = stakedAmounts[_account].add(_amount);
    depositBalances[_account][_depositToken] = depositBalances[_account][_depositToken].add(_amount);
    totalDepositSupply[_depositToken] = totalDepositSupply[_depositToken].add(_amount);

    _mint(_account, _amount);
}
```

StakedQlp.sol

```
function transfer(address _recipient, uint256 _amount) external returns (bool) {
    _transfer(msg.sender, _recipient, _amount);
    return true;
}

function _transfer(address _sender, address _recipient, uint256 _amount) private {
    require(_sender != address(0), "StakedQlp: transfer from the zero address");
    require(_recipient != address(0), "StakedQlp: transfer to the zero address");

    require(
        qlpManager.lastAddedAt(_sender).add(qlpManager.cooldownDuration()) <= block.timestamp,
        "StakedQlp: cooldown duration not yet passed"
    );
    IRewardTracker(feeQlpTracker).unstakeForAccount(_sender, qlp, _amount, _sender);
    IRewardTracker(feeQlpTracker).stakeForAccount(_sender, _recipient, qlp, _amount);
}
```

We would recommend to perhaps loosen the requirements in _validateReceiver, as the same transfer functionality is also implemented in StakedQlp without these requirements (except it being a 1-step instead of 2-step transferal).

Another solution could be to disallow StakedQlp.transfer to pending receivers, but this would increase overhead for transfers by introducing an external call.

QSWP-37. REACTIVATING REWARD TOKEN IMPOSSIBLE WHEN TOTAL LIMIT IS REACHED

SEVERITY: Medium

PATH: RewardDistributor.sol:addRewardToken:L73-95

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The function **addRewardToken** handles the adding and reactivating of reward tokens. There are 2 cases: if a reward token does not yet exist it will be added to the array and activated, if it already existed before, it will only be activated.

The total number of reward tokens in the array is also limited to 10 (MAX_ALL_REWARD_TOKENS).

However, the check for the array's length is done at the very beginning, which means that it assumes to be growing if the function is executed. This is not the case if an older reward token would be reactivated and thus it becomes impossible for a reward token to be reactivated if this limit is ever reached.

Instead, this check should only be done inside of the conditional if the reward is a new token and the array would indeed grow.

```
function addRewardToken(
    address _token
) external override onlyAdmin{
    require(allRewardTokens.length<MAX_ALL_REWARD_TOKENS,"RewardDistributor: too many rewardTokens");

    if (!rewardTokens[_token]) {
        bool isFound;
        uint256 length = allRewardTokens.length;
        for (uint256 i = 0; i < length; i++) {
            if(allRewardTokens[i] == _token){
                isFound = true;
                break;
            }
        }
        if(!isFound){
            allRewardTokens.push(_token);
            allTokens[_token] = true;
        }
        rewardTokens[_token] = true;
        rewardTokenCount++;
    }

}
```

The require(allRewardTokens.length<MAX_ALL_REWARD_TOKENS, "..") check should be moved to the case where the allRewardTokens array is pushed to. E.g.:

```
if(!isFound){
    require(allRewardTokens.length<MAX_ALL_REWARD_TOKENS, "..");
    allRewardTokens.push(_token);
    allTokens[_token] = true;
}
```

QSWP-30. USE != 0 INSTEAD OF > 0 FOR UINT TYPES

SEVERITY: Low

PATH: OrderBook.sol, PositionManager.sol, QlpManager.sol.

REMEDIATION: replace the > 0 checks with != 0 to optimize for gas usage.

STATUS: fixed

DESCRIPTION:

We have identified the following functions in the following contracts that check if the **uint** types are greater than zero using the **> 0** operator:

1. **_amountIn** in **OrderBook.sol:createSwapOrder()** L344-372,
2. **_sizeDelta** in **PositionManager.sol:_validateIncreaseOrder()** L314-352,
3. **size** in **PositionManager.sol:_validatePositionTime()** L353-364,
4. **_amount** in **QlpManager.sol:_addLiquidity()** L209-230,
5. **_qlpAmount** in **QlpManager.sol:_removeLiquidity()** L232-255.

However, using the **!=** operator costs less gas, allowing for gas savings.

1.

```
    function createSwapOrder(
        address[] memory _path,
        uint256 _amountIn,
        uint256 _minOut,
        uint256 _triggerRatio, // tokenB / tokenA
        bool _triggerAboveThreshold,
        uint256 _executionFee,
        bool _shouldWrap,
        bool _shouldUnwrap
    ) external payable nonReentrant {
        require(_path.length == 2 || _path.length == 3, "OB: invalid _path.length");
        require(_path[0] != _path[_path.length - 1], "OB: invalid _path");
        require(_path[0] != usdq && _path[_path.length - 1] != usdq, "OB: invalid token");
        require(_amountIn > 0, "OB: invalid _amountIn");
        require(_executionFee >= minExecutionFee, "OB: insufficient execution fee");

        // always need this call because of mandatory executionFee user has to transfer in ETH
        _transferInETH();

        if (_shouldWrap) {
            require(_path[0] == weth, "OB: only weth could be wrapped");
            require(msg.value == _executionFee.add(_amountIn), "OB: incorrect value transferred");
        } else {
            require(msg.value == _executionFee, "OB: incorrect execution fee transferred");
            IRouter(router).pluginTransfer(_path[0], msg.sender, address(this), _amountIn);
        }

        _createSwapOrder(msg.sender, _path, _amountIn, _minOut, _triggerRatio, _triggerAboveThreshold, _shouldUnwrap,
_executionFee);
    }
```

2.

```
function _validateIncreaseOrder(address _account, uint256 _orderIndex) internal view {
    (
        address _purchaseToken,
        uint256 _purchaseTokenAmount,
        address _collateralToken,
        address _indexToken,
        uint256 _sizeDelta,
        bool _isLong,
        , // triggerPrice
        , // triggerAboveThreshold
        // executionFee
    ) = IOrderBook(orderBook).getIncreaseOrder(_account, _orderIndex);

    _validateMaxGlobalSize(_indexToken, _isLong, _sizeDelta);

    if (!shouldValidateIncreaseOrder) { return; }

    // shorts are okay
    if (!_isLong) { return; }

    // if the position size is not increasing, this is a collateral deposit
    require(_sizeDelta > 0, "PositionManager: long deposit");

    IVault _vault = IVault(vault);
    (uint256 size, uint256 collateral, , , , , ) = _vault.getPosition(_account, _collateralToken, _indexToken, _isLong);

    // if there is no existing position, do not charge a fee
    if (size == 0) { return; }

    uint256 nextSize = size.add(_sizeDelta);
    uint256 collateralDelta = _vault.tokenToUsdMin(_purchaseToken, _purchaseTokenAmount);
    uint256 nextCollateral = collateral.add(collateralDelta);

    uint256 prevLeverage = size.mul(BASIS_POINTS_DIVISOR).div(collateral);
    // allow for a maximum of a increasePositionBufferBps decrease since there might be some swap fees taken from the collateral
    uint256 nextLeverageWithBuffer = nextSize.mul(BASIS_POINTS_DIVISOR + increasePositionBufferBps).div(nextCollateral);

    require(nextLeverageWithBuffer >= prevLeverage, "PositionManager: long leverage decrease");
}
```

3.

```
    function _validatePositionTime(
        address _account,
        address _collateralToken,
        address _indexToken,
        bool _isLong
    ) private view{
        IVault _vault = IVault(vault);
        (uint256 size, , , , , , , uint lastIncreasedTime) = _vault.getPosition(_account, _collateralToken, _indexToken, _isLong);
        require(size > 0, "PositionManager: empty position");
        uint256 minDelayTime = partnerMinStayingOpenTime[_account]>0 ? partnerMinStayingOpenTime[_account] :
minStayingOpenTime;
        require(lastIncreasedTime.add(minDelayTime) <= block.timestamp, "PositionManager: min delay not yet passed");
    }
```

4.

```
    function _addLiquidity(address _fundingAccount, address _account, address _token, uint256 _amount, uint256
_minUsdq, uint256 _minQlp) private returns (uint256) {
        require(_amount > 0, "QlpManager: invalid _amount");

        // calculate aum before buyUSDQ
        uint256 aumInUsdq = getAumInUsdq(true);
        uint256 qlpSupply = IERC20(qlp).totalSupply();

        IERC20(_token).safeTransferFrom(_fundingAccount, address(vault), _amount);
        uint256 usdqAmount = vault.buyUSDQ(_token, address(this));
        require(usdqAmount >= _minUsdq, "QlpManager: insufficient USDQ output");

        uint256 mintAmount = aumInUsdq == 0 ? usdqAmount : usdqAmount.mul(qlpSupply).div(aumInUsdq);
        require(mintAmount >= _minQlp, "QlpManager: insufficient QLP output");

        IMintable(qlp).mint(_account, mintAmount);

        lastAddedAt[_account] = block.timestamp;

        emit AddLiquidity(_account, _token, _amount, aumInUsdq, qlpSupply, usdqAmount, mintAmount);

        return mintAmount;
    }
```

5.

```
    function _removeLiquidity(address _account, address _tokenOut, uint256 _qlpAmount, uint256 _minOut, address
_receiver) private returns (uint256) {
        require(_qlpAmount > 0, "QlpManager: invalid _qlpAmount");
        require(lastAddedAt[_account].add(cooldownDuration) <= block.timestamp, "QlpManager: cooldown duration not
yet passed");

        // calculate aum before sellUSDQ
        uint256 aumInUsdq = getAumInUsdq(false);
        uint256 qlpSupply = IERC20(qlp).totalSupply();

        uint256 usdqAmount = _qlpAmount.mul(aumInUsdq).div(qlpSupply);
        uint256 usdqBalance = IERC20(usdq).balanceOf(address(this));
        if (usdqAmount > usdqBalance) {
            IUSDQ(usdq).mint(address(this), usdqAmount.sub(usdqBalance));
        }

        IMintable(qlp).burn(_account, _qlpAmount);

        IERC20(usdq).transfer(address(vault), usdqAmount);
        uint256 amountOut = vault.sellUSDQ(_tokenOut, _receiver);
        require(amountOut >= _minOut, "QlpManager: insufficient output");

        emit RemoveLiquidity(_account, _tokenOut, _qlpAmount, aumInUsdq, qlpSupply, usdqAmount, amountOut);

        return amountOut;
    }
```

QSWP-38. ADD REWARD TOKEN OPTIMISATION

SEVERITY: Low

PATH: RewardDistributor.sol:addRewardToken:L73-95

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The function **addRewardToken** makes use of a for-loop to check if a token ever existed before pushing it to the **allRewardTokens** array.

However, this is unnecessary because the **allTokens** mapping is only every set when the reward token is pushed to array. The array never shrinks and the mapping is never set to false, so it can be used to lookup the existence in constant time.

```
function addRewardToken(
    address _token
) external override onlyAdmin{
    require(allRewardTokens.length<MAX_ALL_REWARD_TOKENS,"RewardDistributor: too many rewardTokens");

    if (!rewardTokens[_token]) {
        bool isFound;
        uint256 length = allRewardTokens.length;
        for (uint256 i = 0; i < length; i++) {
            if(allRewardTokens[i] == _token){
                isFound = true;
                break;
            }
        }
        if(!isFound){
            allRewardTokens.push(_token);
            allTokens[_token] = true;
        }
        rewardTokens[_token] = true;
        rewardTokenCount++;
    }

}
```

The addRewardToken should be optimised by removing the for-loop and make use of the allTokens mapping.

For example (and by also taking the recommendation of issue QSWP-37: Reactivating reward token impossible when total limit is reached into account):

```
function addRewardToken(
    address _token
) external override onlyAdmin{
    if (!rewardTokens[_token]) {
        if (!alltokens[_token]) {
            require(allRewardTokens.length < MAX_ALL_REWARD_TOKENS,"RewardDistributor: too many rewardTokens");
            allRewardTokens.push(_token);
            allTokens[_token] = true;
        }
        rewardTokens[_token] = true;
        rewardTokenCount++;
    }
}
```

QSWP-29. USING CALLDATA INSTEAD OF MEMORY

SEVERITY: Informational

PATH: BasePositionManager.sol:setMaxGlobalSizes:L134-146

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The parameters of the function **setMaxGlobalSizes** can be marked as **calldata** parameters instead of **memory** in favour of gas saving.

```
function setMaxGlobalSizes(
    address[] memory _tokens,
    uint256[] memory _longSizes,
    uint256[] memory _shortSizes
) external onlyAdmin {
    for (uint256 i = 0; i < _tokens.length; i++) {
        address token = _tokens[i];
        maxGlobalLongSizes[token] = _longSizes[i];
        maxGlobalShortSizes[token] = _shortSizes[i];
    }

    emit SetMaxGlobalSizes(_tokens, _longSizes, _shortSizes);
}
```

QSWP-35. UNUSED IMPORT OF REENTRANCYGUARD

SEVERITY: Informational

PATH: StakedQlp.sol, RewardDistributor.sol

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

We found that the ReentrancyGuard library is imported and implemented by a number of contracts. However, the modifier **nonReentrant** is not used by some of the contracts:

1. StakedQlp.sol
2. RewardDistributor.sol

The ReentrancyGuard import should be removed.

```
import "../libraries/utils/ReentrancyGuard.sol";
```

QSWP-36. LACK OF EVENTS

SEVERITY: Informational

PATH: RewardDistributor.sol:addRewardToken, removeRewardToken (L73-95, L97-105)

REMEDIATION: add and emits events for adding and removing reward tokens

STATUS: fixed

DESCRIPTION:

This update adds new functionality to the RewardDistributor, such as the addition of multiple reward tokens.

The new functions that add and remove reward tokens update the state, however they do not emit events to notify off-chain watchers of changes.

```
function addRewardToken(
    address _token
) external override onlyAdmin{
    require(allRewardTokens.length<MAX_ALL_REWARD_TOKENS,"RewardDistributor: too many
rewardTokens");

    if (!rewardTokens[_token]) {
        bool isFound;
        uint256 length = allRewardTokens.length;
        for (uint256 i = 0; i < length; i++) {
            if(allRewardTokens[i] == _token){
                isFound = true;
                break;
            }
        }
        if(!isFound){
            allRewardTokens.push(_token);
            allTokens[_token] = true;
        }
        rewardTokens[_token] = true;
        rewardTokenCount++;
    }

}

function removeRewardToken(
    address _token
) external override onlyAdmin{
    if(rewardTokens[_token]){
        require(tokensPerInterval[_token] == 0,"RewardDistributor: tokensPerInterval must be zero");
        delete rewardTokens[_token];
        rewardTokenCount--;
    }
}
```