SMART CONTRACT AUDIT REPORT

for

Limit Order

December 11, 2024
Prepared by: Securly3

Document Properties

Client	SquadSwap
Title	Smart Contract Audit Report for Limit Order
Target	Limit Order
Version	1.0
Author	Securly3
Auditors	Securly3
Reviewed by	Securly3
Approved by	Securly3
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 10, 2024	Securly3	Final Release

Contents

1 | Introduction

The `SquadLimitOrder` smart contract provides a decentralized mechanism for creating and managing limit orders for token pairs in a liquidity pool. It incorporates functionality for token transfers, minting liquidity positions, and emergency operations. The audit covers security, business logic, and adherence to best practices.

2 | Methodology

The audit follows these steps:

1. **Static Analysis**: Reviewing the source code for patterns of vulnerabilities using manual and automated tools.
2. **Semantic Consistency Checks**: Ensuring alignment between the contract's intended functionality and actual implementation.
3. **Business Logic Analysis**: Examining critical features like order execution, fee mechanisms, and access control.
4. **Risk Evaluation**: Categorizing vulnerabilities using the OWASP Risk Rating Methodology.

Severity is categorized as follows:

Likelihood/Impact	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Informational

3 | Findings

3.1 Summary of Issues

This section provides a summary of vulnerabilities identified in the `SquadLimitOrder` contract, categorized by severity. The findings highlight critical security risks, inefficiencies, and best practice violations, with recommendations provided for each issue.

Severity	Count
Critical	0
High	2 🟥🟥
Medium	4 🟨🟨🟨🟨
Low	4 🟪🟪🟪🟪
Informational	1 🟦

3.2 Key Findings

The table below outlines the key issues discovered during the audit, categorized by severity and status. High-severity findings require urgent action, while medium and low-severity findings suggest improvements to enhance security and functionality.

ID	Severity	Title	Status
SLO-001	High	Unauthorized Access to executeLimitOrder	Confirmed
SLO-002	High	Reentrancy Risk	Confirmed
SLO-003	Medium	Gas Refund Logic Vulnerability	Confirmed
SLO-004	Medium	Lack of Validation on Token Transfers	Confirmed
SLO-005	Medium	Inadequate Whitelist Management	Confirmed

SLO-006	Medium	Hardcoded Deadline in mintPosition	Confirmed
SLO-007	Low	Potential Reentrancy in `reduceLiquidityAndCollect`	Resolved
SLO-008	Low	Unchecked External Calls in `withdrawToken`	Confirmed
SLO-009	Low	Gas Limit Risks in Batch Operations	Confirmed
SLO-010	Low	Implicit Assumptions in State Updates	Confirmed
SLO-011	Informational	Inefficient Token Approval in `handleTokenTransfersAndApproval`	Confirmed

4 | Detailed Findings

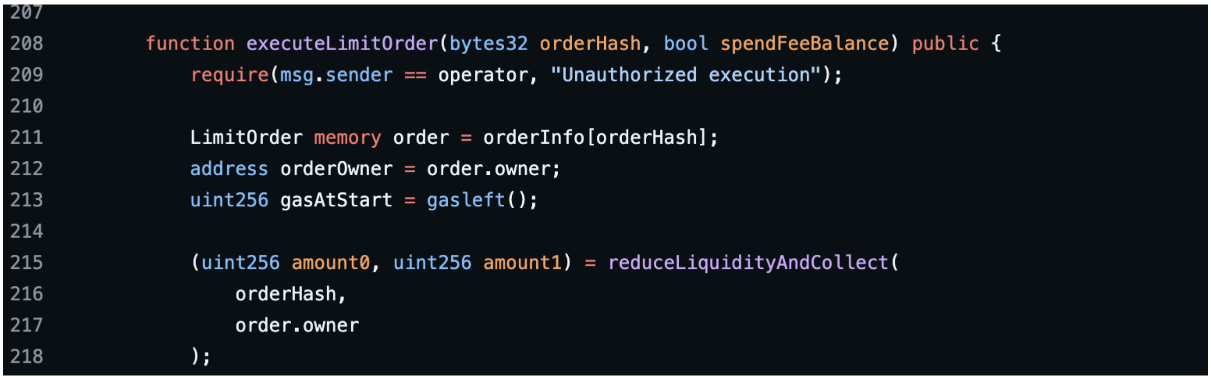
4.1: Unauthorized Access to `executeLimitOrder`

ID: SLO-001

Severity: High

Description:
The `executeLimitOrder` function is protected by a check that ensures only the `operator` can call it. However, the `operator` address can be updated arbitrarily by the owner, which creates a risk of malicious actors being granted control over this critical function.

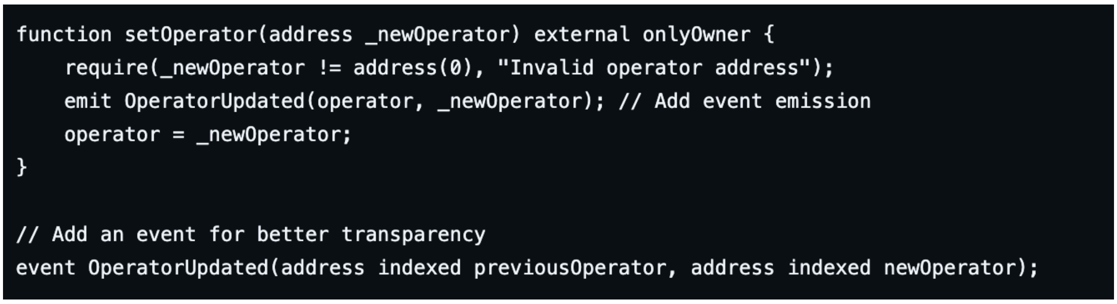
Relevant Code:



```
    function executeLimitOrder(bytes32 orderHash, bool spendFeeBalance) public {
        require(msg.sender == operator, "Unauthorized execution");

        LimitOrder memory order = orderInfo[orderHash];
        address orderOwner = order.owner;
        uint256 gasAtStart = gasleft();

        (uint256 amount0, uint256 amount1) = reduceLiquidityAndCollect(
            orderHash,
            order.owner
        );
```

Recommendation:

Introduce a multisig or governance mechanism for setting the operator. Additionally, emit an event when the operator is changed and add access control for this update.



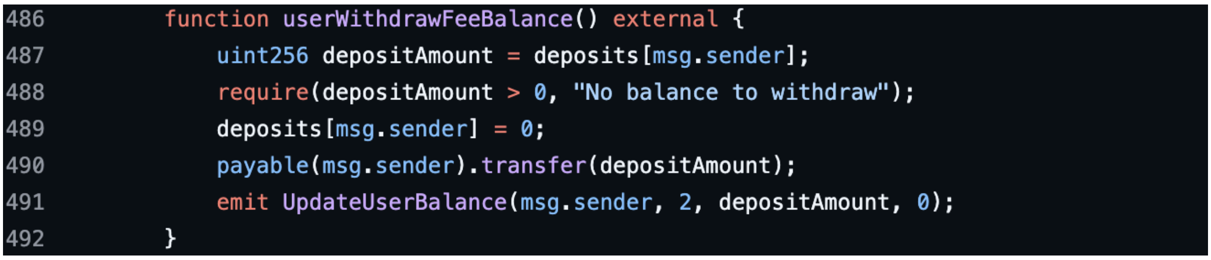
```
function setOperator(address _newOperator) external onlyOwner {
    require(_newOperator != address(0), "Invalid operator address");
    emit OperatorUpdated(operator, _newOperator); // Add event emission
    operator = _newOperator;
}

// Add an event for better transparency
event OperatorUpdated(address indexed previousOperator, address indexed newOperator);
```

4.2 Reentrancy Risk

ID: SLO-002

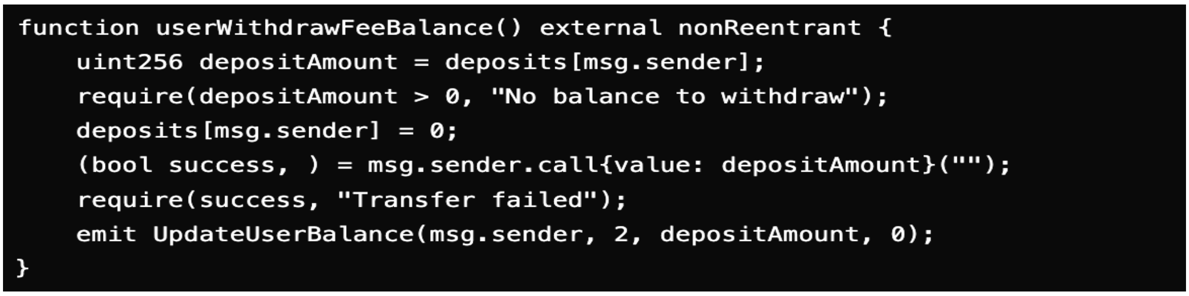
Severity: High

Description:
Functions like userWithdrawFeeBalance and withdrawETH involve ETH transfers to users without implementing the Checks-Effects-Interactions pattern. This can expose the contract to reentrancy attacks.



```
    function userWithdrawFeeBalance() external {
        uint256 depositAmount = deposits[msg.sender];
        require(depositAmount > 0, "No balance to withdraw");
        deposits[msg.sender] = 0;
        payable(msg.sender).transfer(depositAmount);
        emit UpdateUserBalance(msg.sender, 2, depositAmount, 0);
    }
```

Recommendation:
Apply OpenZeppelin's ReentrancyGuard to prevent reentrant calls. Update the function as follows:



```
function userWithdrawFeeBalance() external nonReentrant {
    uint256 depositAmount = deposits[msg.sender];
    require(depositAmount > 0, "No balance to withdraw");
    deposits[msg.sender] = 0;
    (bool success, ) = msg.sender.call{value: depositAmount}("");
    require(success, "Transfer failed");
    emit UpdateUserBalance(msg.sender, 2, depositAmount, 0);
}
```

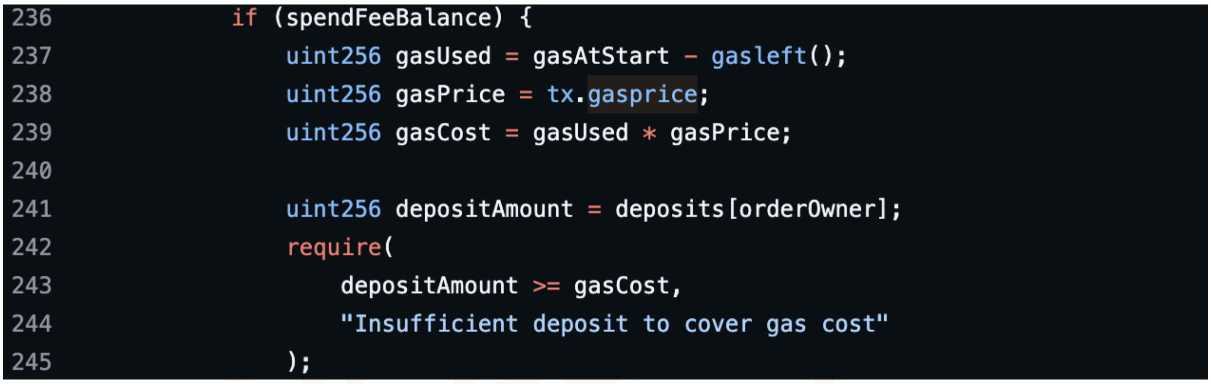
4.3 Gas Refund Logic Vulnerability

ID: SLO-003

Severity: Medium

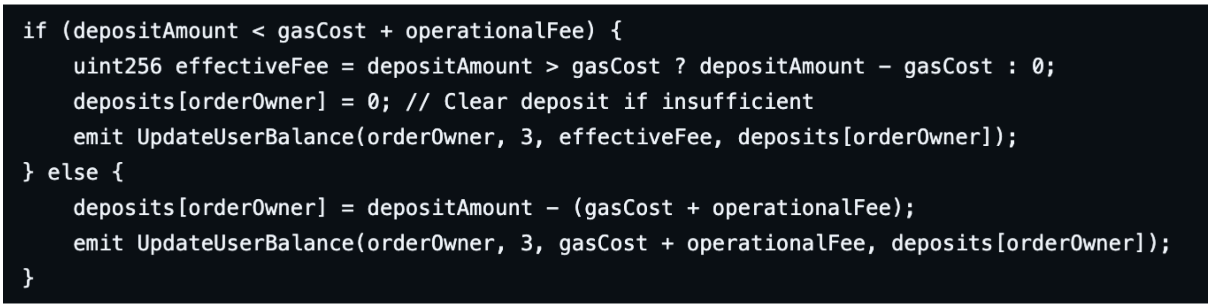
Description:
In the `executeLimitOrder` function, the gas cost is refunded using the depositor's balance. If the deposit is insufficient, the function reverts, potentially creating a denial-of-service (DoS) scenario.

Relevant Code:



```
if (spendFeeBalance) {
    uint256 gasUsed = gasAtStart - gasleft();
    uint256 gasPrice = tx.gasprice;
    uint256 gasCost = gasUsed * gasPrice;

    uint256 depositAmount = deposits[orderOwner];
    require(
        depositAmount >= gasCost,
        "Insufficient deposit to cover gas cost"
    );
```

Recommendation:
Add a fallback mechanism where operational fees are reduced if the user's balance is insufficient.



```
if (depositAmount < gasCost + operationalFee) {
    uint256 effectiveFee = depositAmount > gasCost ? depositAmount - gasCost : 0;
    deposits[orderOwner] = 0; // Clear deposit if insufficient
    emit UpdateUserBalance(orderOwner, 3, effectiveFee, deposits[orderOwner]);
} else {
    deposits[orderOwner] = depositAmount - (gasCost + operationalFee);
    emit UpdateUserBalance(orderOwner, 3, gasCost + operationalFee, deposits[orderOwner]);
}
```

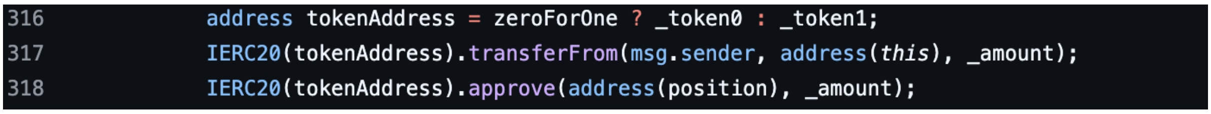
4.4 Lack of Validation on Token Transfers

ID: SLO-004

Severity: Medium

Description:
The `handleTokenTransfersAndApproval` function does not validate the success of `transferFrom` or `approve` calls. This could lead to silent failures if a token does not behave as expected.

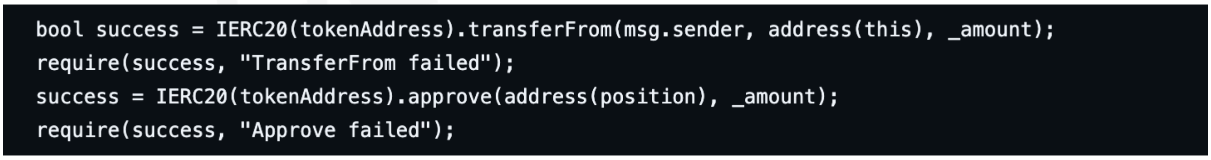
Relevant Code:



```
            address tokenAddress = zeroForOne ? _token0 : _token1;
            IERC20(tokenAddress).transferFrom(msg.sender, address(this), _amount);
            IERC20(tokenAddress).approve(address(position), _amount);
```

Recommendation:

Verify the success of the `transferFrom` and `approve` calls:



```
bool success = IERC20(tokenAddress).transferFrom(msg.sender, address(this), _amount);
require(success, "TransferFrom failed");
success = IERC20(tokenAddress).approve(address(position), _amount);
require(success, "Approve failed");
```

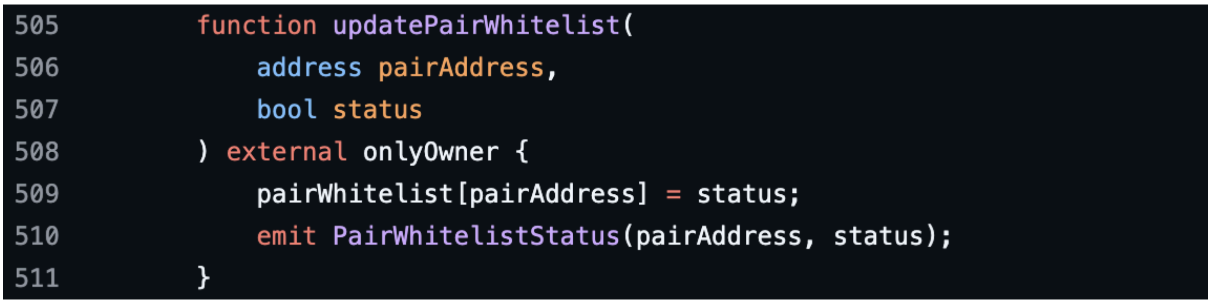
4.5 Inadequate Whitelist Management

ID: SLO-005

Severity: Medium

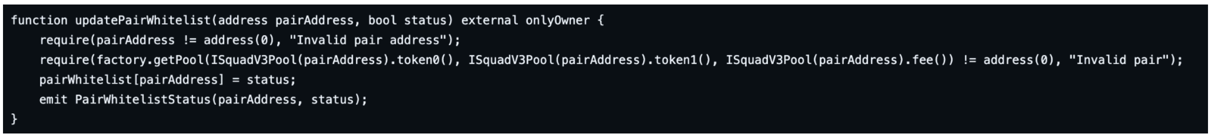
Description:
The `updatePairWhitelist` function allows any pair address to be whitelisted without validation. This introduces risks if a malicious or incompatible pair is added.

Relevant Code:



```
        function updatePairWhitelist(
            address pairAddress,
            bool status
        ) external onlyOwner {
            pairWhitelist[pairAddress] = status;
            emit PairWhitelistStatus(pairAddress, status);
        }
```

Recommendation:
Add checks to validate the pair address, ensuring it conforms to the expected contract interface.



```
function updatePairWhitelist(address pairAddress, bool status) external onlyOwner {
    require(pairAddress != address(0), "Invalid pair address");
    require(factory.getPool(ISquadV3Pool(pairAddress).token0(), ISquadV3Pool(pairAddress).token1(), ISquadV3Pool(pairAddress).fee()) != address(0), "Invalid pair");
    pairWhitelist[pairAddress] = status;
    emit PairWhitelistStatus(pairAddress, status);
}
```

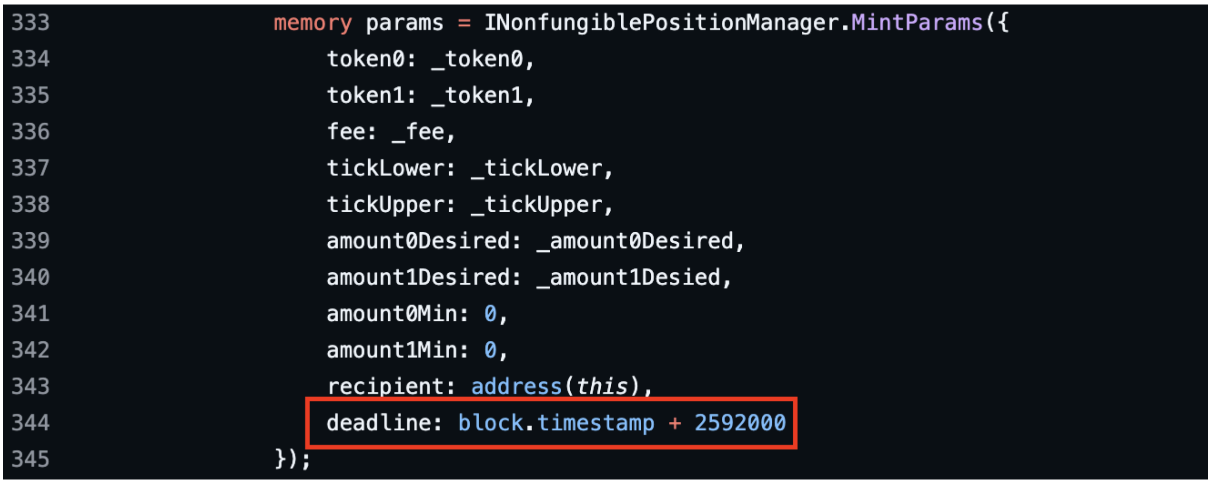
4.6 Hardcoded Deadline in mintPosition

ID: SLO-006

Severity: Medium

Description:
The `mintPosition` function hardcodes a 30-day deadline (`block.timestamp + 2592000`). This reduces flexibility in varying use cases.

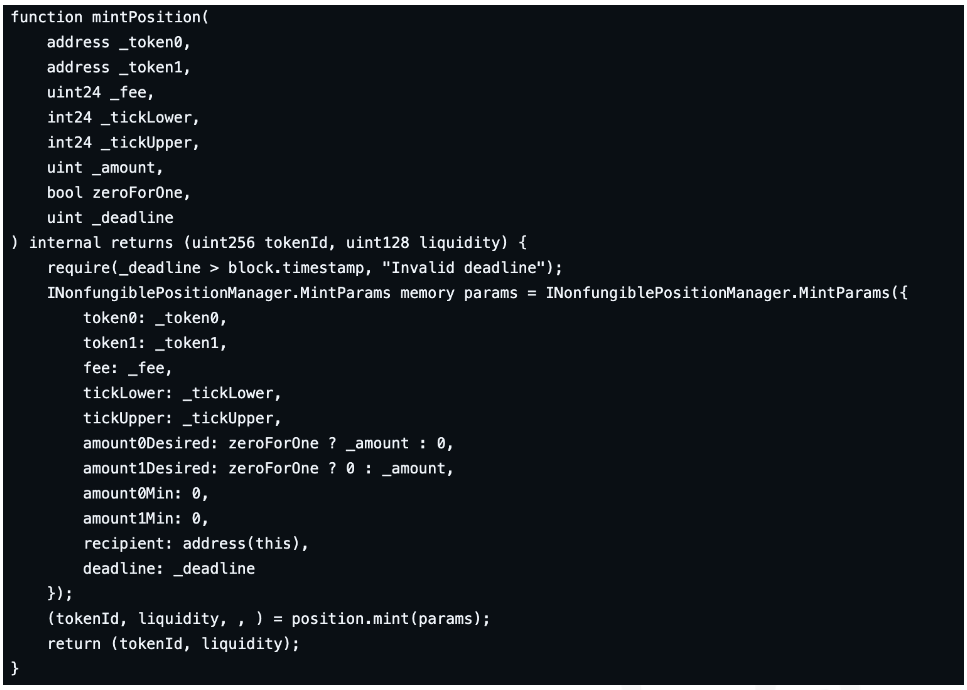
Relevant Code:



```
                memory params = INonfungiblePositionManager.MintParams({
                    token0: _token0,
                    token1: _token1,
                    fee: _fee,
                    tickLower: _tickLower,
                    tickUpper: _tickUpper,
                    amount0Desired: _amount0Desired,
                    amount1Desired: _amount1Desied,
                    amount0Min: 0,
                    amount1Min: 0,
                    recipient: address(this),
                    deadline: block.timestamp + 2592000
                });
```

Recommendation:

Allow users to specify a deadline as a parameter:



```
function mintPosition(
    address _token0,
    address _token1,
    uint24 _fee,
    int24 _tickLower,
    int24 _tickUpper,
    uint _amount,
    bool zeroForOne,
    uint _deadline
) internal returns (uint256 tokenId, uint128 liquidity) {
    require(_deadline > block.timestamp, "Invalid deadline");
    INonfungiblePositionManager.MintParams memory params = INonfungiblePositionManager.MintParams({
        token0: _token0,
        token1: _token1,
        fee: _fee,
        tickLower: _tickLower,
        tickUpper: _tickUpper,
        amount0Desired: zeroForOne ? _amount : 0,
        amount1Desired: zeroForOne ? 0 : _amount,
        amount0Min: 0,
        amount1Min: 0,
        recipient: address(this),
        deadline: _deadline
    });
    (tokenId, liquidity, , ) = position.mint(params);
    return (tokenId, liquidity);
}
```

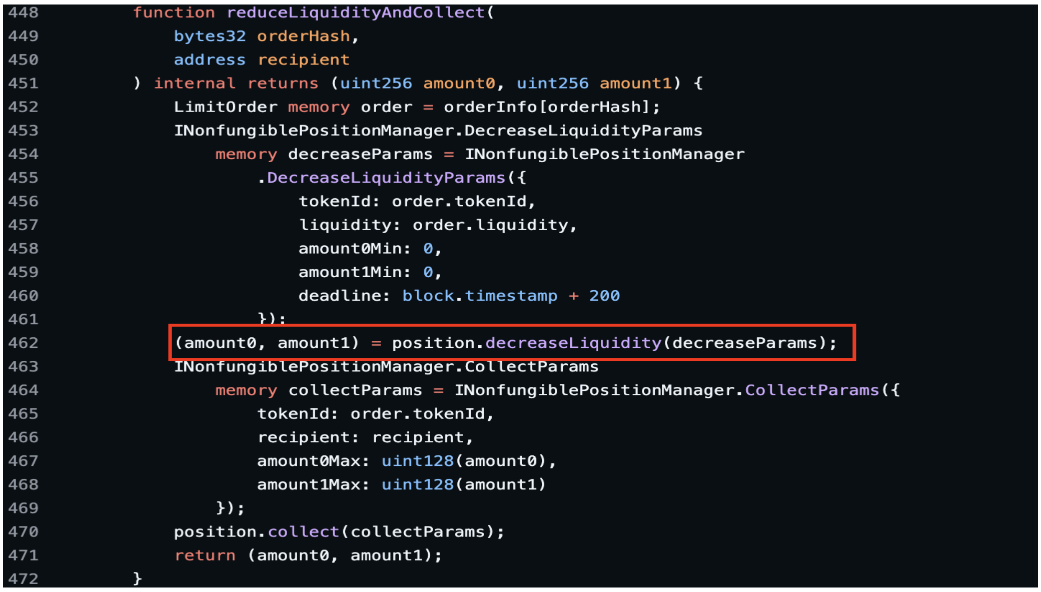
4.7 Potential Reentrancy in `reduceLiquidityAndCollect`

ID: SLO-007

Severity: Low

Description:
The `reduceLiquidityAndCollect` function interacts with external contracts (`position.decreaseLiquidity` and `position.collect`) without following the `Checks-Effects-Interactions` pattern, potentially exposing it to reentrancy attacks.

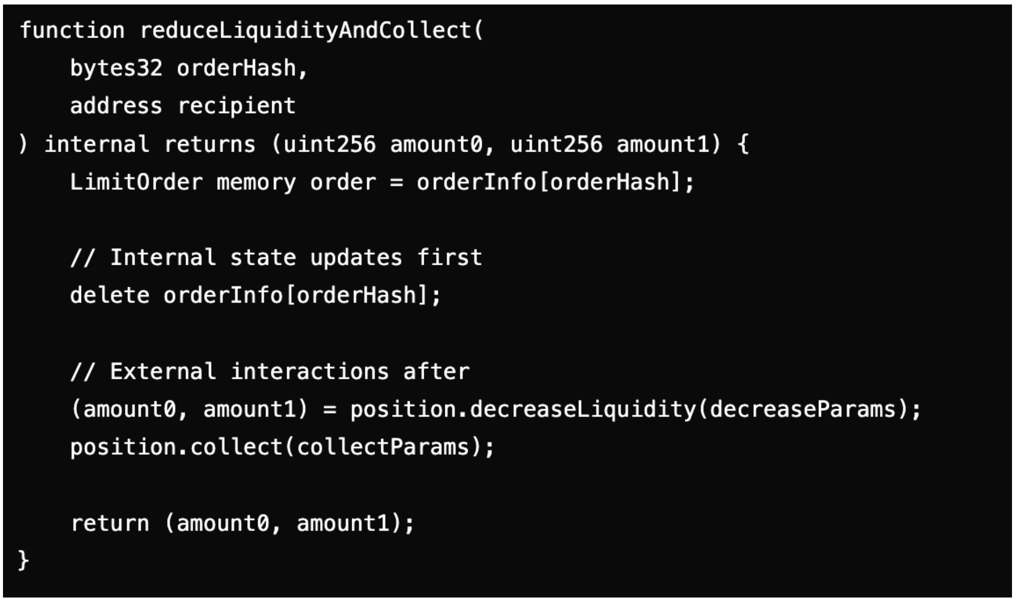
Relevant Code:



```
    function reduceLiquidityAndCollect(
        bytes32 orderHash,
        address recipient
    ) internal returns (uint256 amount0, uint256 amount1) {
        LimitOrder memory order = orderInfo[orderHash];
        INonfungiblePositionManager.DecreaseLiquidityParams
            memory decreaseParams = INonfungiblePositionManager
                .DecreaseLiquidityParams({
                    tokenId: order.tokenId,
                    liquidity: order.liquidity,
                    amount0Min: 0,
                    amount1Min: 0,
                    deadline: block.timestamp + 200
                });
        (amount0, amount1) = position.decreaseLiquidity(decreaseParams);
        INonfungiblePositionManager.CollectParams
            memory collectParams = INonfungiblePositionManager.CollectParams({
                tokenId: order.tokenId,
                recipient: recipient,
                amount0Max: uint128(amount0),
                amount1Max: uint128(amount1)
            });
        position.collect(collectParams);
        return (amount0, amount1);
    }
```

Recommendation:

Use the `Checks-Effects-Interactions` pattern to ensure that internal state updates are made before any external calls.



```
function reduceLiquidityAndCollect(
    bytes32 orderHash,
    address recipient
) internal returns (uint256 amount0, uint256 amount1) {
    LimitOrder memory order = orderInfo[orderHash];

    // Internal state updates first
    delete orderInfo[orderHash];

    // External interactions after
    (amount0, amount1) = position.decreaseLiquidity(decreaseParams);
    position.collect(collectParams);

    return (amount0, amount1);
}
```

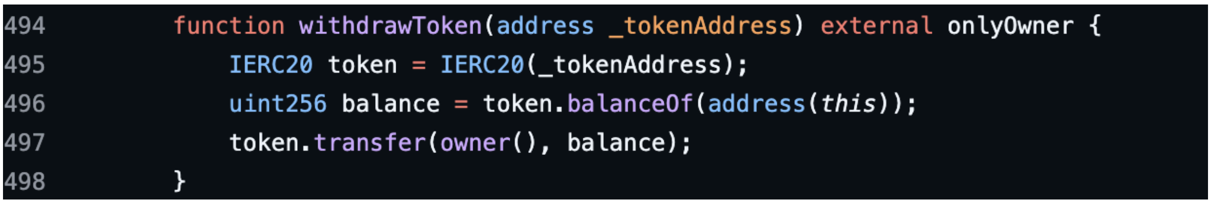
4.8 Unchecked External Calls in `withdrawToken`

ID: SLO-008

Severity: Low

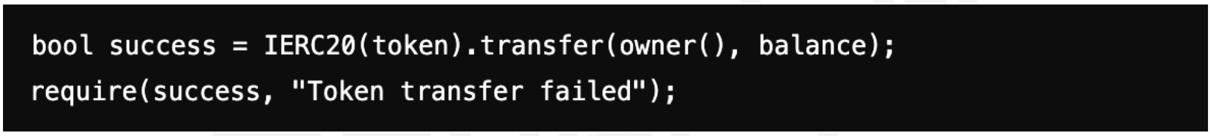
Description:
The `withdrawToken` function uses `IERC20.transfer` but does not verify its return value, which could result in silent failures.

Relevant Code:



```
    function withdrawToken(address _tokenAddress) external onlyOwner {
        IERC20 token = IERC20(_tokenAddress);
        uint256 balance = token.balanceOf(address(this));
        token.transfer(owner(), balance);
    }
```

Recommendation:
Validate the return value of the transfer to ensure the operation succeeds.



```
bool success = IERC20(token).transfer(owner(), balance);
require(success, "Token transfer failed");
```

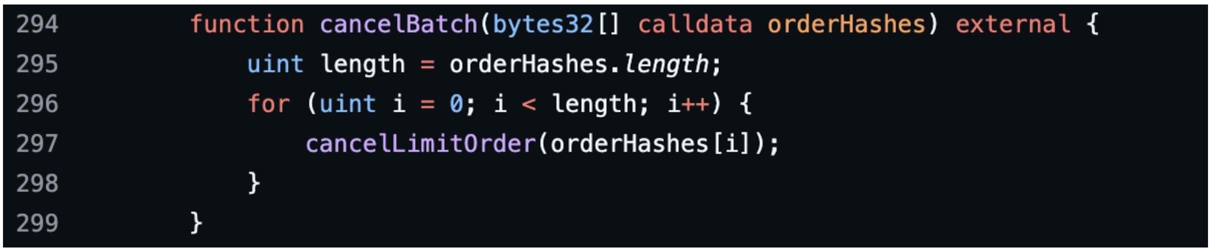
4.9 Gas Limit Risks in Batch Operations

ID: SLO-009

Severity: Low

Description:
The `cancelBatch` function iterates over an arbitrary-sized array, which may cause out-of-gas errors**.**

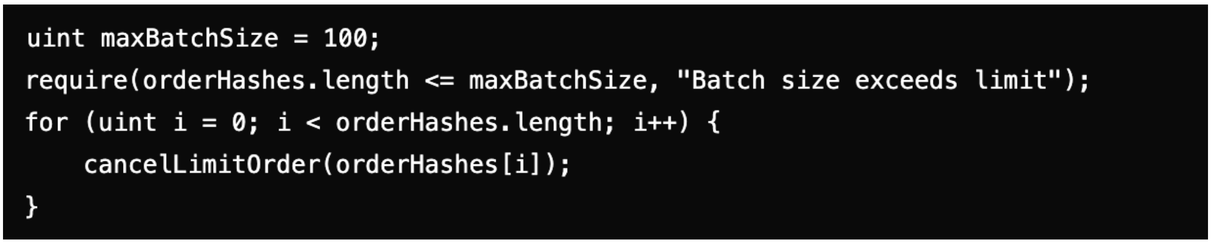
Relevant Code:



```
    function cancelBatch(bytes32[] calldata orderHashes) external {
        uint length = orderHashes.length;
        for (uint i = 0; i < length; i++) {
            cancelLimitOrder(orderHashes[i]);
        }
    }
```

Recommendation:

Introduce a limit on batch size



```
uint maxBatchSize = 100;
require(orderHashes.length <= maxBatchSize, "Batch size exceeds limit");
for (uint i = 0; i < orderHashes.length; i++) {
    cancelLimitOrder(orderHashes[i]);
}
```

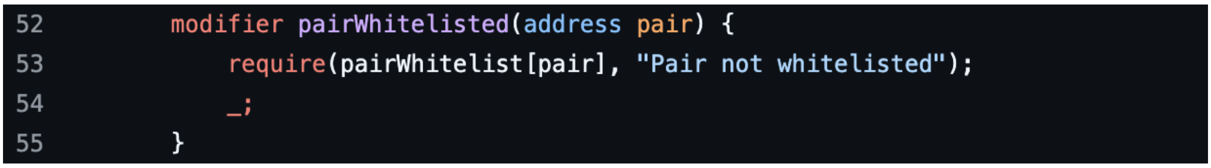
4.10 Implicit Assumptions in State Updates

ID: SLO-010

Severity: Low

Description:
Functions like `makeLimitOrder` assume that `pairWhitelist` is always accurate and consistent.

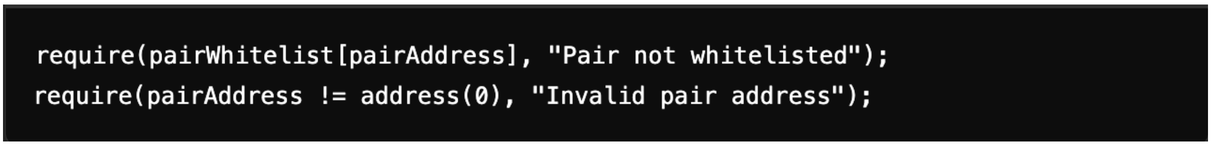
Relevant Code:



```
        modifier pairWhitelisted(address pair) {
            require(pairWhitelist[pair], "Pair not whitelisted");
            _;
        }
```

Recommendation:

Include additional integrity checks for `pairWhitelist`:



```
require(pairWhitelist[pairAddress], "Pair not whitelisted");
require(pairAddress != address(0), "Invalid pair address");
```

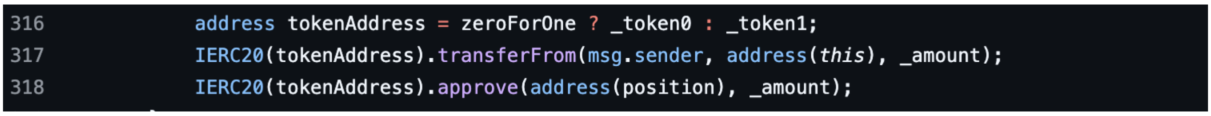
4.11 Inefficient Token Approval in `handleTokenTransfersAndApproval`

ID: SLO-011

Severity: Informational

Description:
The function renews approvals on every transfer, increasing gas costs unnecessarily.

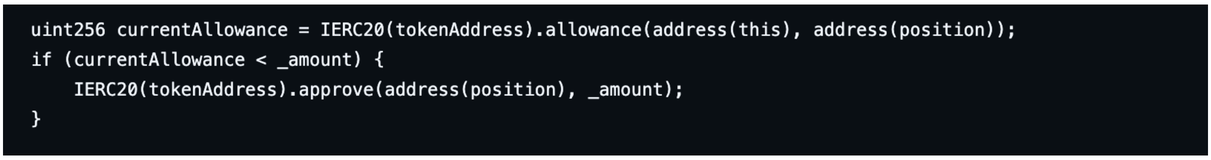
Relevant Code:



```
        address tokenAddress = zeroForOne ? _token0 : _token1;
        IERC20(tokenAddress).transferFrom(msg.sender, address(this), _amount);
        IERC20(tokenAddress).approve(address(position), _amount);
```

Recommendation:

Only renew approvals when the current allowance is insufficient.



```
uint256 currentAllowance = IERC20(tokenAddress).allowance(address(this), address(position));
if (currentAllowance < _amount) {
    IERC20(tokenAddress).approve(address(position), _amount);
}
```

5 | Conclusion

The `SquadLimitOrder` contract is well-structured and adheres to modern Solidity standards. However, several issues related to access control, logic validation, and gas efficiency were identified. By implementing the recommended changes, the contract will become significantly more secure and efficient.

6 | References

1. OWASP Risk Rating Methodology:

   https://cwe.mitre.org/data/definitions/ 190.html.

2. OpenZeppelin Security Best Practices:
   https://cwe.mitre.org/data/definitions/287.html.

3. Common Weaknes Enumeration:

   https://cwe.mitre.org/ data/definitions/628.html.