BLOCKSEC

Security Audit
Report for stove-protocol-contracts

Date: January 13, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Habittrade
Target	stove-protocol-contracts

Version History

Version	Date	Description
1.0	January 13, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-01-14 07:08:50+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of stove-protocol-contracts of StoveProtocol.

The protocol establishes an on-chain stock trading market that enables the issuance and trading of tokenized stocks. Stock tokens are created via a factory contract, while certain stock-related operations, such as dividends, stock splits, and reverse splits are handled through trusted off-chain logic and executed on-chain by privileged operators. User trading activity, including buying and selling stock tokens, is facilitated through the RFQSettlement contract, which manages order execution and settlement. By combining on-chain enforcement with off-chain computation, the protocol aims to provide a flexible and efficient framework for tokenized equity management and trading.

Note this audit only focuses on the smart contracts in the following directories/files:

- src/*

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (Version 1), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (Version 0), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
stove-protocol-contracts	Version 1	001713ff5055f02d07764e2dcae0f5c53abae9d1
	Version 2	487edd88d922b1e8456ff30ce849f26c67dccecf

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset.

[1] https://github.com/StoveProtocol/stove-protocol-contracts

Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

- Access control
- Permission management
- Whitelist and blacklist mechanisms
- Initialization consistency
- Improper use of the proxy system
- Reentrancy
- Denial of Service (DoS)
- Untrusted external call and control flow
- Exception handling
- Data handling and flow
- Events operation
- Error-prone randomness
- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism

- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

		Likelihood	
		High	*Low*
Impact	*High*	High	Medium
	Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **ten** potential security issues. Besides, we have **two** recommendations and **two** notes.

- High Risk: 1
- Medium Risk: 6
- Low Risk: 3
- Recommendation: 2
- Note: 2

ID	Severity	Description	Category	Status
1	High	Lack of check in functions `permitTransferFrom()` and `permitTransferFromBatch()`	Security Issue	Fixed
2	Medium	Incorrect lock caller context in batch order filling	Security Issue	Fixed
3	Medium	Lack of management routing in `StockTokenManager`	Security Issue	Fixed
4	Medium	Incomplete delisting coverage in stock token management	Security Issue	Fixed
5	Medium	Potential loss of funds on expired order relocking	Security Issue	Fixed
6	Medium	Incorrect `stockToken` update logic	Security Issue	Confirmed
7	Medium	Lack of principal address validation in function `lockOrder()`	Security Issue	Fixed
8	Low	Inconsistent incentive allocation in order filling	Security Issue	Confirmed
9	Low	Inconsistent `exchange` validation in function `updateStockInfo()`	Security Issue	Fixed
10	Low	Inconsistent lock window configuration	Security Issue	Fixed
11	-	Improper logic in function `computeAddress()`	Recommendation	Fixed
12	-	Inconsistent event emission in `invalidateNonce()` and `invalidateNonces()`	Recommendation	Fixed
13	-	Potential centralization risks	Note	-
14	-	Trusted off-chain logic	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lack of check in functions `permitTransferFrom()` and `permitTransferFromBatch()`

Severity High

Status Fixed in `Version 2`

Introduced by Version 1

Description The Permit2 contract enables signature-based token transfers, where a user signs a permit authorizing a transfer and a third party can submit the transaction accordingly. In both functions permitTransferFrom() and permitTransferFromBatch(), the signed permit includes a spender field, which suggests that only a specific spender is intended to execute the transfer. However, the current implementation does not validate that msg.sender matches permit.spender. As a result, attackers can front-run the transactions with a legally signed permit and replace the transferDetails.to their controlled address to gain profit.

```
function permitTransferFrom(
    PermitTransferFrom calldata permit,
    SignatureTransferDetails calldata transferDetails,
    address owner,
    bytes calldata signature
) external {
    // Validate deadline
    if (block.timestamp > permit.deadline) revert SignatureExpired();

    // Validate nonce
    if (nonces[owner][permit.nonce]) revert InvalidNonce();
    if (permit.nonce < currentNonce[owner]) revert InvalidNonce();

    // Validate amount
    if (transferDetails.requestedAmount > permit.permitted.amount) revert InsufficientAllowance();

    // Verify signature
    bytes32 permitHash = _hashPermitTransferFrom(permit);
    bytes32 digest = _hashTypedDataV4(permitHash);
    address recoveredAddress = digest.recover(signature);

    if (recoveredAddress != owner) revert InvalidSignature();

    // Mark nonce as used
    nonces[owner][permit.nonce] = true;

    // Execute transfer
    IERC20(permit.permitted.token).safeTransferFrom(owner, transferDetails.to, transferDetails.requestedAmount);

    emit Permit(owner, permit.permitted.token, permit.spender, transferDetails.requestedAmount, permit.deadline);
}
```

Listing 2.1: src/Permit2.sol

```
function permitTransferFromBatch(
    PermitTransferFrom calldata permit,
    SignatureTransferDetails[] calldata transferDetails,
    address owner,
    bytes calldata signature
) external {
    // Validate deadline
    if (block.timestamp > permit.deadline) revert SignatureExpired();

    // Validate nonce
    if (nonces[owner][permit.nonce]) revert InvalidNonce();
    if (permit.nonce < currentNonce[owner]) revert InvalidNonce();

    // Verify signature
    bytes32 permitHash = _hashPermitTransferFrom(permit);
    bytes32 digest = _hashTypedDataV4(permitHash);
    address recoveredAddress = digest.recover(signature);

    if (recoveredAddress != owner) revert InvalidSignature();

    // Mark nonce as used
    nonces[owner][permit.nonce] = true;

    // Execute transfers
    uint256 totalAmount;
    for (uint256 i = 0; i < transferDetails.length; i++) {
        totalAmount += transferDetails[i].requestedAmount;
        IERC20(permit.permitted.token).safeTransferFrom(
            owner, transferDetails[i].to, transferDetails[i].requestedAmount
        );
    }

    // Validate total amount
    if (totalAmount > permit.permitted.amount) revert InsufficientAllowance();

    emit Permit(owner, permit.permitted.token, permit.spender, totalAmount, permit.deadline);
}
```

Listing 2.2: src/Permit2.sol

Impact The user who signed the permit may lose funds.

Suggestion Add a check to ensure that the caller matches the specified `spender`.

2.1.2 Incorrect lock caller context in batch order filling

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The protocol requires an order to be locked by a taker before it can be filled, and this relationship is enforced in function `_checkLocks()`, which allows filling only when `msg.sender` equals `lock.locker`.

In the batch filling flow, function `batchFillOrders()` fills each order via an external self-call to function `_fillOrderInternal()`. As a result, when function `_checkLocks()` is executed, `msg.sender` becomes the `RFQSettlement` contract itself rather than the original taker who locked the order.

Since `lock.locker` is recorded as the taker address, the caller check in function `_checkLocks()` fails, and the order is treated as being locked by another party. This causes the fill to revert, preventing batch order filling from succeeding even when the order was properly locked by the taker.

```
function batchFillOrders(SignedOrder[] calldata signedOrders, uint256[] calldata fillQuantities, uint256[] calldata takerIncentives, bytes32[] calldata expectedOrderHashes, string[] calldata brokerOrderIds)
    external
    onlyTaker
    whenNotPaused
    nonReentrant
    returns (BatchFillResult[] memory results)
{
    require(signedOrders.length == fillQuantities.length, "Length mismatch");
    require(signedOrders.length == takerIncentives.length, "Incentives length mismatch");
    require(signedOrders.length == expectedOrderHashes.length, "Hash length mismatch");
    require(signedOrders.length == brokerOrderIds.length, "BrokerOrderIds length mismatch");

    address taker = msg.sender;
    results = new BatchFillResult[](signedOrders.length);

    for (uint256 i = 0; i < signedOrders.length; i++) {
        try this._fillOrderInternal(signedOrders[i], fillQuantities[i], takerIncentives[i], taker, expectedOrderHashes[i], brokerOrderIds[i]) returns (FillResult memory result) {
            results[i] = BatchFillResult({success: true, result: result, errorData: ""});
        } catch (bytes memory errorData) {
            results[i] = BatchFillResult({
                success: false,
                result: FillResult({filledQuantity: 0, filledAmount: 0, feeAmount: 0, incentiveAmount: 0}),
                errorData: errorData
            });
        }
    }
}
```

Listing 2.3: src/RFQSettlement.sol

```
function _fillOrderInternal(SignedOrder calldata signedOrder, uint256 fillQuantity, uint256 takerIncentive, address taker, bytes32 expectedOrderHash, string calldata brokerOrderId)
    external
    returns (FillResult memory result)
{
    require(msg.sender == address(this), "Internal only");
    return _fillOrderLogic(signedOrder, fillQuantity, takerIncentive, taker, expectedOrderHash, brokerOrderId);
}
```

Listing 2.4: src/RFQSettlement.sol

```
function _fillOrderLogic(SignedOrder calldata signedOrder, uint256 fillQuantity, uint256 takerIncentive, address taker, bytes32 expectedOrderHash, string calldata brokerOrderId)
```

```
        internal
        returns (FillResult memory result)
    {
        Order calldata order = signedOrder.order;

        _validateOrder(order);

        bytes32 orderHash = _hashOrder(order);
        if (orderHash != expectedOrderHash) revert OrderHashMismatch();
        _verifySignature(orderHash, order.maker, signedOrder.signature);

        if (_checkLocks(orderHash)) {
            revert OrderLocked();
        }

        // Check if order is locked (required for escrow-based settlement)
        OrderLock storage lock = orderLocks[orderHash];
        if (lock.locker == address(0)) {
            revert NotLocked(); // Must lock order first to escrow assets
        }

        uint256 currentFill = fills[orderHash];
        if (currentFill + fillQuantity > order.quantity) revert InsufficientQuantity();

        // Calculate remaining available incentive
        uint256 remainingQuantity = order.quantity - currentFill;
        uint256 remainingIncentive = (order.incentive * remainingQuantity) / order.quantity;

        // Validate taker incentive does not exceed remaining available incentive
        if (takerIncentive > remainingIncentive) revert ExcessiveIncentive();

        // Note: fillQuantity is integer (stock tokens have 0 decimals)
        uint256 fillAmount = order.price * fillQuantity;
        uint256 feeAmount = (fillAmount * feeRate) / BASIS_POINTS;
        uint256 incentiveAmount = takerIncentive; // Use taker-specified incentive
        uint256 netAmount = fillAmount - feeAmount;

        fills[orderHash] = currentFill + fillQuantity;

        address stockToken = order.isBuy
            ? manager.getStockToken(order.ticker, order.exchange)
            : manager.getExistingStockToken(order.ticker, order.exchange);
        if (stockToken == address(0)) revert InvalidTicker();

        IERC20 assetToken = IERC20(order.asset);
        address takerPrincipal = getTakerPrincipal(taker);

        if (order.isBuy) {
            // Buy order: Release escrowed AssetToken and mint StockToken
            manager.mintTokensForUser(stockToken, order.principal, fillQuantity, orderHash);

            // Transfer from escrow (contract) to taker and fee recipient
            assetToken.safeTransfer(takerPrincipal, netAmount);
            if (feeAmount > 0) {
                assetToken.safeTransfer(feeRecipient, feeAmount);
            }
            if (incentiveAmount > 0) {
                assetToken.safeTransfer(takerPrincipal, incentiveAmount);
            }
        } else {
            // Sell order: Burn escrowed StockToken and transfer AssetToken from taker

            // Burn from contract’s escrow, not from principal
            manager.burnTokensFromUser(stockToken, address(this), fillQuantity, orderHash);

            // Taker pays principal (this is not from escrow)
            assetToken.safeTransferFrom(takerPrincipal, order.principal, netAmount);
            if (feeAmount > 0) {
                assetToken.safeTransferFrom(takerPrincipal, feeRecipient, feeAmount);
            }
            // Incentive is released from escrow
            if (incentiveAmount > 0) {
                assetToken.safeTransfer(takerPrincipal, incentiveAmount);
            }
        }

        // Update lock state: deduct filled amounts from escrow
        lock.remainingQuantity -= fillQuantity;

        if (order.isBuy) {
            // Buy order: deduct fillAmount + incentive from escrow
            // Note: feeAmount is already deducted from fillAmount when calculating netAmount,
            // so we only deduct the gross amount (fillAmount) + incentive
            lock.escrowedAssetAmount -= (fillAmount + incentiveAmount);
        } else {
            // Sell order: deduct stock and incentive from escrow
            lock.escrowedStockAmount -= fillQuantity;
            lock.escrowedAssetAmount -= incentiveAmount;
        }

        // If fully filled, delete the lock
        if (lock.remainingQuantity == 0) {
            delete orderLocks[orderHash];
        }

        result = FillResult({
            filledQuantity: fillQuantity,
            filledAmount: fillAmount,
            feeAmount: feeAmount,
            incentiveAmount: incentiveAmount
        });

        emit OrderFilled(
            orderHash,
            order.maker,
            taker,
            order.ticker,
            order.exchange,
            order.isBuy,
            fillQuantity,
            fillAmount,
            feeAmount,
            incentiveAmount,
            brokerOrderId
        );
    }
```

Listing 2.5: src/RFQSettlement.sol

```
function _checkLocks(bytes32 orderHash) internal view returns (bool isLocked) {
    OrderLock storage lock = orderLocks[orderHash];

    // locker == address(0) means unlocked or cancelled
    if (lock.locker == address(0)) {
        return false; // Not locked, can fill
    }

    // Check if lock is expired
    if (block.timestamp >= lock.lockedAt + lockWindow) {
        revert LockExpired(); // Must call cancelLock first to refund escrow
    }

    // Check if caller is the locker
    if (msg.sender == lock.locker) {
        return false; // Locker can fill their own locked order
    }

    return true; // Locked by someone else
}
```

Listing 2.6: src/RFQSettlement.sol

Impact The `batchFillOrders()` function cannot be executed successfully, making batch order filling unusable.

Suggestion Revise the logic accordingly.

2.1.3 Lack of management routing in `StockTokenManager`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `Vault` contract relies on a timelocked, two step process to manage privileged configuration, including updating the manager role and other administrative parameters. All such operations are restricted to the current `Vault` manager. Since contract `StockTokenManager` is designated as the `Vault` manager, it is expected to act as the execution layer for these administrative actions.

However, contract StockTokenManager does not implement any logic to invoke the corresponding management functions on the contract Vault, including manager updates, batched withdrawals, timelock delay and expiry configuration, or proposal cancellation. This omission prevents these management actions from being initiated or completed, effectively blocking the contract Vault's administrative control flow.

```solidity
/**
 * @notice Propose to update the manager address (step 1)
 * @param newManager New manager address
 * @return proposalId The proposal ID
 * @dev Only current manager can propose
 */
function proposeSetManager(address newManager) external onlyManager returns (bytes32 proposalId) {
    if (newManager == address(0)) revert ZeroAddress();

    bytes32 actionHash = _hashAction(this.executeSetManager.selector, abi.encode(newManager));
    proposalId = _createProposal(actionHash);
}

/**
 * @notice Execute the manager update after timelock delay (step 2)
 * @param proposalId The proposal ID from proposeSetManager
 * @param newManager New manager address (must match proposal)
 * @dev Only current manager can execute
 */
function executeSetManager(bytes32 proposalId, address newManager) external onlyManager {
    if (newManager == address(0)) revert ZeroAddress();

    bytes32 actionHash = _hashAction(this.executeSetManager.selector, abi.encode(newManager));
    _executeProposal(proposalId, actionHash);

    address oldManager = manager;
    manager = newManager;

    emit ManagerUpdated(oldManager, newManager);
}
```

Listing 2.7: src/Vault.sol

Impact The contract Vault is unable to perform critical administrative updates, leading to a loss of governance and operational control.

Suggestion Add explicit management functions in contract StockTokenManager to correctly forward and execute all required administrative actions for the contract Vault.

2.1.4 Incomplete delisting coverage in stock token management

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description The delisting mechanism relies on function `triggerDelisting()` to both mark a stock token as `Delisted` and register a pending `Delisting` action for a specified user in `userPendingActions`.

However, once the token status is updated from `Normal` to `Delisted`, subsequent calls to function `triggerDelisting()` will be rejected as the check requires the status to be `Normal`. As a result, the delisting logic can only be executed once, and all affected users must be explicitly specified and recorded at that time.

If any holder is omitted during the initial delisting trigger, there is no way to add that user later, since the function cannot be called again after the token is delisted. This leads to incomplete delisting coverage, where some token holders are never registered for the `Delisting` action.

```
function triggerDelisting(address token, uint256 actionId, address user)
    external
    onlyOperator
    whenNotPaused
{
    if (!isRegisteredToken[token]) revert TokenNotRegistered();
    if (IStockToken(token).status() != uint8(TokenStatus.Normal)) revert TokenAlreadyDelisted();

    // Check if action already exists
    SimplifiedCorporateAction storage existingAction = globalActions[token][actionId];
    if (existingAction.timestamp != 0) {
        // Action exists, verify type matches
        if (existingAction.actionType != CorporateActionType.Delisting) revert ActionTypeMismatch();
        // Idempotent: action already registered with same type, skip registration
    } else {
        // Create new action
        globalActions[token][actionId] = SimplifiedCorporateAction({
            actionType: CorporateActionType.Delisting,
            timestamp: block.timestamp,
            triggeredBy: msg.sender
        });

        // Update token status only when first creating the action
        IStockToken(token).setStatus(uint8(TokenStatus.Delisted));
    }

    // Add to user's pending actions (check for duplicates first)
    uint256[] storage pending = userPendingActions[token][user];
    bool alreadyPending = false;
    for (uint256 i = 0; i < pending.length; i++) {
        if (pending[i] == actionId) {
            alreadyPending = true;
            break;
        }
    }

    if (!alreadyPending) {
        pending.push(actionId);
    }

    emit CorporateActionTriggered(token, actionId, user, CorporateActionType.Delisting, block.timestamp);
}
```

Listing 2.8: src/StockTokenManager.sol

Impact Some `stockToken` holders may not receive delisting compensation, resulting in potential loss of funds.

Suggestion Revise the logic accordingly.

2.1.5 Potential loss of funds on expired order relocking

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description An expired order can be relocked without refunding or reconciling assets escrowed under the previous lock. During relocking, function `lockOrder()` recalculates the escrow amounts solely based on the original order parameters and the computed `availableQuantity`, without accounting for assets that were already escrowed or partially consumed by prior fills.

In the case of partial fills, any remaining escrowed funds from the previous lock are not refunded to the maker. Instead, the relocking process transfers additional assets from the maker and overwrites the existing lock state, including `remainingQuantity`, `escrowedAssetAmount`, and `escrowedStockAmount`.

As a result, previously escrowed funds may become permanently unaccounted for, while additional assets are transferred into the contract under the assumption of a fresh lock. Furthermore, prior fill-related accounting is effectively overridden by the new lock state, leading to incorrect escrow balances and potential loss of funds.

```
     * @dev Escrows assets from principal to contract
     * @dev Supports re-locking after cancelLock or partial fills
     */
    function lockOrder(SignedOrder calldata signedOrder, bytes32 expectedOrderHash) external onlyTaker whenNotPaused nonReentrant {
        Order calldata order = signedOrder.order;

        // Validate order first
        _validateOrder(order);

        // Verify signature
        bytes32 orderHash = _hashOrder(order);
        if (orderHash != expectedOrderHash) revert OrderHashMismatch();
        _verifySignature(orderHash, order.maker, signedOrder.signature);

        OrderLock storage lock = orderLocks[orderHash];
        // Check if already locked by someone (locker != address(0) means locked)
        if (lock.locker != address(0)) {
            // Check if lock is still valid (not expired)
            if (block.timestamp < lock.lockedAt + lockWindow) {
                revert OrderAlreadyLocked();
            }
            // Lock expired, can be re-locked
        }

        // Calculate available quantity to lock
        uint256 availableQuantity;
        if (lock.remainingQuantity > 0) {
            // This order has been partially filled and lock was cancelled
            // Use the remaining quantity stored in lock
            availableQuantity = lock.remainingQuantity;
        } else {
            // New order or fully filled order, check total fills
            uint256 currentFill = fills[orderHash];
            availableQuantity = order.quantity - currentFill;
        }

        if (availableQuantity == 0) revert InsufficientQuantity();

        // Calculate escrow amounts
        uint256 escrowAsset;
        uint256 escrowStock;
        uint256 remainingIncentive = (order.incentive * availableQuantity) / order.quantity;

        if (order.isBuy) {
            // Buy order: escrow AssetToken (price * qty + incentive)
            // Note: quantity is integer (stock tokens have 0 decimals)
            escrowAsset = (order.price * availableQuantity) + remainingIncentive;
            escrowStock = 0;
        } else {
            // Sell order: escrow StockToken + incentive in AssetToken
            escrowStock = availableQuantity;
            escrowAsset = remainingIncentive;
        }

        // Transfer assets to contract for escrow
        IERC20 assetToken = IERC20(order.asset);

        if (order.isBuy) {
            // Buy order: transfer AssetToken from principal
            assetToken.safeTransferFrom(order.principal, address(this), escrowAsset);
        } else {
            // Sell order: transfer StockToken from principal
            address stockToken = manager.getExistingStockToken(order.ticker, order.exchange);
            if (stockToken == address(0)) revert InvalidTicker();

            IERC20(stockToken).safeTransferFrom(order.principal, address(this), escrowStock);

            // Also transfer incentive if exists
            if (escrowAsset > 0) {
                assetToken.safeTransferFrom(order.principal, address(this), escrowAsset);
            }
        }

        // Update lock state with escrow info
        lock.locker = msg.sender;
        lock.lockedAt = block.timestamp;
        lock.remainingQuantity = availableQuantity;
        lock.escrowedAssetAmount = escrowAsset;
        lock.escrowedStockAmount = escrowStock;

        emit OrderLockCreated(orderHash, order.maker, order.nonce, msg.sender);
    }
```

Listing 2.9: src/RFQSettlement.sol

Impact Order makers may lose both previously escrowed assets and newly transferred funds.

Suggestion Require explicit cancellation and refund of any existing lock before relocking, and ensure escrow calculations correctly account for partial fills before transferring or updating escrow state.

2.1.6 Incorrect `stockToken` update logic

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The order execution and escrow logic assumes that a stock token can be consistently resolved from the factory registry using a key derived from the order's `ticker` and `exchange` throughout the entire lifecycle of an order, including locking, filling, and refunding.

However, several privileged functions can modify stock token metadata or factory mappings without coordinating with existing orders and locks.

First, function `processTickerChangeDirect()` can update the factory's `ticker` mapping while there are still active buy orders that have already been locked but not yet filled. When such an order is later executed, the original `ticker` and `exchange` recorded in the order may no longer resolve to the intended stock token. In this case, function `getStockToken()` may deploy a new stock token and mint it to the user, even though the order was locked against a different token instance.

In addition, function `updateStockInfo()` allows the `exchange` field of a stock token to be modified without updating the factory registry. This can further desynchronize the token's metadata from the factory mapping used during order execution.

These inconsistencies can also affect refund paths such as function `cancelLock()`, where the contract attempts to resolve an existing stock token using the order's `ticker` and `exchange`. If the resolution fails, escrowed stock tokens may not be refunded correctly.

```
function updateStockInfo(address token, string calldata companyName, string calldata exchange)
```

```
        external
        onlyMinter
        whenNotPaused
    {
        if (!isRegisteredToken[token]) revert TokenNotRegistered();
        IStockToken(token).updateStockInfo(companyName, exchange);
    }
```

Listing 2.10: src/StockTokenManager.sol

```
    function updateStockInfo(string memory _companyName, string memory _exchange) external onlyOwner {
        _stockInfo.companyName = _companyName;
        _stockInfo.exchange = _exchange;
        emit StockInfoUpdated(_stockInfo.ticker, _companyName, _exchange);
    }
```

Listing 2.11: src/StockToken.sol

```
    function cancelLock(SignedOrder calldata signedOrder) external nonReentrant {
        Order calldata order = signedOrder.order;
        bytes32 orderHash = _hashOrder(order);

        OrderLock storage lock = orderLocks[orderHash];

        // Check if there's an active lock
        if (lock.locker == address(0)) revert NotLocked();

        // Check permission
        bool isExpired = block.timestamp >= lock.lockedAt + lockWindow;
        if (!isExpired) {
            // Before expiry, only locker can cancel
            require(msg.sender == lock.locker, "Only locker can cancel before expiry");
        }
        // After expiry, anyone can cancel

        // Record amounts before refunding
        uint256 remainingQty = lock.remainingQuantity;
        uint256 refundAsset = lock.escrowedAssetAmount;
        uint256 refundStock = lock.escrowedStockAmount;
        uint256 totalRefund = refundAsset + refundStock;

        // Refund escrowed assets to principal
        IERC20 assetToken = IERC20(order.asset);

        if (order.isBuy) {
            // Buy order: refund AssetToken (includes remaining price payment + incentive)
            if (refundAsset > 0) {
                assetToken.safeTransfer(order.principal, refundAsset);
            }
        } else {
            // Sell order: refund StockToken + incentive
            if (refundStock > 0) {
```

```
            address stockToken = manager.getExistingStockToken(order.ticker, order.exchange);
            if (stockToken != address(0)) {
                IERC20(stockToken).safeTransfer(order.principal, refundStock);
            }
        }
        if (refundAsset > 0) {
            assetToken.safeTransfer(order.principal, refundAsset);
        }
    }

    // Decide whether to delete or clear lock
    if (remainingQty == 0) {
        // Order fully filled during lock, delete the record
        delete orderLocks[orderHash];
    } else {
        // Order has remaining quantity, clear lock but keep remainingQuantity
        lock.locker = address(0);
        lock.lockedAt = 0;
        lock.escrowedAssetAmount = 0;
        lock.escrowedStockAmount = 0;
        // lock.remainingQuantity stays unchanged for potential re-locking
    }

    emit OrderLockCancelled(orderHash, remainingQty, totalRefund);
}
```

Listing 2.12: src/RFQSettlement.sol

```
function _fillOrderLogic(SignedOrder calldata signedOrder, uint256 fillQuantity, uint256 takerIncentive, address taker, bytes32 expectedOrderHash, string calldata brokerOrderId)
    internal
    returns (FillResult memory result)
{
    Order calldata order = signedOrder.order;

    _validateOrder(order);

    bytes32 orderHash = _hashOrder(order);
    if (orderHash != expectedOrderHash) revert OrderHashMismatch();
    _verifySignature(orderHash, order.maker, signedOrder.signature);

    if (_checkLocks(orderHash)) {
        revert OrderLocked();
    }

    // Check if order is locked (required for escrow-based settlement)
    OrderLock storage lock = orderLocks[orderHash];
    if (lock.locker == address(0)) {
        revert NotLocked(); // Must lock order first to escrow assets
    }

    uint256 currentFill = fills[orderHash];
    if (currentFill + fillQuantity > order.quantity) revert InsufficientQuantity();
```

```
        // Calculate remaining available incentive
        uint256 remainingQuantity = order.quantity - currentFill;
        uint256 remainingIncentive = (order.incentive * remainingQuantity) / order.quantity;

        // Validate taker incentive does not exceed remaining available incentive
        if (takerIncentive > remainingIncentive) revert ExcessiveIncentive();

        // Note: fillQuantity is integer (stock tokens have 0 decimals)
        uint256 fillAmount = order.price * fillQuantity;
        uint256 feeAmount = (fillAmount * feeRate) / BASIS_POINTS;
        uint256 incentiveAmount = takerIncentive; // Use taker-specified incentive
        uint256 netAmount = fillAmount - feeAmount;

        fills[orderHash] = currentFill + fillQuantity;

        address stockToken = order.isBuy
            ? manager.getStockToken(order.ticker, order.exchange)
            : manager.getExistingStockToken(order.ticker, order.exchange);
        if (stockToken == address(0)) revert InvalidTicker();

        IERC20 assetToken = IERC20(order.asset);
        address takerPrincipal = getTakerPrincipal(taker);

        if (order.isBuy) {
            // Buy order: Release escrowed AssetToken and mint StockToken
            manager.mintTokensForUser(stockToken, order.principal, fillQuantity, orderHash);

            // Transfer from escrow (contract) to taker and fee recipient
            assetToken.safeTransfer(takerPrincipal, netAmount);
            if (feeAmount > 0) {
                assetToken.safeTransfer(feeRecipient, feeAmount);
            }
            if (incentiveAmount > 0) {
                assetToken.safeTransfer(takerPrincipal, incentiveAmount);
            }
        } else {
            // Sell order: Burn escrowed StockToken and transfer AssetToken from taker

            // Burn from contract's escrow, not from principal
            manager.burnTokensFromUser(stockToken, address(this), fillQuantity, orderHash);

            // Taker pays principal (this is not from escrow)
            assetToken.safeTransferFrom(takerPrincipal, order.principal, netAmount);
            if (feeAmount > 0) {
                assetToken.safeTransferFrom(takerPrincipal, feeRecipient, feeAmount);
            }
            // Incentive is released from escrow
            if (incentiveAmount > 0) {
                assetToken.safeTransfer(takerPrincipal, incentiveAmount);
            }
        }
        // Update lock state: deduct filled amounts from escrow
        lock.remainingQuantity -= fillQuantity;

        if (order.isBuy) {
            // Buy order: deduct fillAmount + incentive from escrow
            // Note: feeAmount is already deducted from fillAmount when calculating netAmount,
            // so we only deduct the gross amount (fillAmount) + incentive
            lock.escrowedAssetAmount -= (fillAmount + incentiveAmount);
        } else {
            // Sell order: deduct stock and incentive from escrow
            lock.escrowedStockAmount -= fillQuantity;
            lock.escrowedAssetAmount -= incentiveAmount;
        }

        // If fully filled, delete the lock
        if (lock.remainingQuantity == 0) {
            delete orderLocks[orderHash];
        }

        result = FillResult({
            filledQuantity: fillQuantity,
            filledAmount: fillAmount,
            feeAmount: feeAmount,
            incentiveAmount: incentiveAmount
        });

        emit OrderFilled(
            orderHash,
            order.maker,
            taker,
            order.ticker,
            order.exchange,
            order.isBuy,
            fillQuantity,
            fillAmount,
            feeAmount,
            incentiveAmount,
            brokerOrderId
        );
    }
```

Listing 2.13: src/RFQSettlement.sol

Impact `StockToken` misconfiguration may lead to inconsistent state and failed order processing or refunds.

Suggestion Revise the logic accordingly.

Feedback from the project It can be checked off-chain before triggering corporate actions. If detected, the order will be manually unlocked or rejected to ensure consistency.

2.1.7 Lack of principal address validation in function `lockOrder()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `lockOrder()` locks a signed order and escrows the corresponding `assetToken` or `stockToken` from the order's `principal`. The function verifies that the order is signed by the `maker` but does not enforce that the `maker` is the owner of the `principal` address. As a result, a malicious `maker` can specify any arbitrary `principal` who has approved the contract, causing the function `lockOrder()`to transfer tokens from that `principal` into escrow. This allows order creation without the `maker` bearing any cost and gives the `maker` unauthorized access to others' funds.

```
function lockOrder(SignedOrder calldata signedOrder, bytes32 expectedOrderHash) external
    onlyTaker whenNotPaused nonReentrant {
    Order calldata order = signedOrder.order;

    // Validate order first
    _validateOrder(order);

    // Verify signature
    bytes32 orderHash = _hashOrder(order);
    if (orderHash != expectedOrderHash) revert OrderHashMismatch();
    _verifySignature(orderHash, order.maker, signedOrder.signature);

    OrderLock storage lock = orderLocks[orderHash];

    // Check if already locked by someone (locker != address(0) means locked)
    if (lock.locker != address(0)) {
        // Check if lock is still valid (not expired)
        if (block.timestamp < lock.lockedAt + lockWindow) {
            revert OrderAlreadyLocked();
        }
        // Lock expired, can be re-locked
    }

    // Calculate available quantity to lock
    uint256 availableQuantity;
    if (lock.remainingQuantity > 0) {
        // This order has been partially filled and lock was cancelled
        // Use the remaining quantity stored in lock
        availableQuantity = lock.remainingQuantity;
    } else {
        // New order or fully filled order, check total fills
        uint256 currentFill = fills[orderHash];
        availableQuantity = order.quantity - currentFill;
    }

    if (availableQuantity == 0) revert InsufficientQuantity();

    // Calculate escrow amounts
    uint256 escrowAsset;
    uint256 escrowStock;
    uint256 remainingIncentive = (order.incentive * availableQuantity) / order.quantity;

    if (order.isBuy) {
        // Buy order: escrow AssetToken (price * qty + incentive)
        // Note: quantity is integer (stock tokens have 0 decimals)
        escrowAsset = (order.price * availableQuantity) + remainingIncentive;
        escrowStock = 0;
    } else {
        // Sell order: escrow StockToken + incentive in AssetToken
        escrowStock = availableQuantity;
        escrowAsset = remainingIncentive;
    }

    // Transfer assets to contract for escrow
    IERC20 assetToken = IERC20(order.asset);

    if (order.isBuy) {
        // Buy order: transfer AssetToken from principal
        assetToken.safeTransferFrom(order.principal, address(this), escrowAsset);
    } else {
        // Sell order: transfer StockToken from principal
        address stockToken = manager.getExistingStockToken(order.ticker, order.exchange);
        if (stockToken == address(0)) revert InvalidTicker();

        IERC20(stockToken).safeTransferFrom(order.principal, address(this), escrowStock);

        // Also transfer incentive if exists
        if (escrowAsset > 0) {
            assetToken.safeTransferFrom(order.principal, address(this), escrowAsset);
        }
    }

    // Update lock state with escrow info
    lock.locker = msg.sender;
    lock.lockedAt = block.timestamp;
    lock.remainingQuantity = availableQuantity;
    lock.escrowedAssetAmount = escrowAsset;
    lock.escrowedStockAmount = escrowStock;

    emit OrderLockCreated(orderHash, order.maker, order.nonce, msg.sender);
}
```

Listing 2.14: src/RFQSettlement.sol

Impact A malicious `maker` can create orders using assets from arbitrary users who have approved the contract.

Suggestion Enforce binding between `maker` and `principal` in function `lockOrder()` to ensure the escrowed funds belong to the signing `maker`.

2.1.8 Inconsistent incentive allocation in order filling

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The `_fillOrderLogic()` function allows a `taker` to specify an incentive amount when executing a fill, which is validated only against the remaining available incentive for the order. The incentive value is not constrained by the actual filled quantity, allowing incentive extraction that is not aligned with the proportion of the order being filled. This creates a mismatch between fill execution and incentive distribution.

```
        // Calculate remaining available incentive
        uint256 remainingQuantity = order.quantity - currentFill;
        uint256 remainingIncentive = (order.incentive * remainingQuantity) / order.quantity;

        // Validate taker incentive does not exceed remaining available incentive
        if (takerIncentive > remainingIncentive) revert ExcessiveIncentive();
```

Listing 2.15: src/RFQSettlement.sol

Impact Order creators may experience unintended incentive depletion, resulting in potential loss of funds.

Suggestion Constrain incentive allocation to be proportional to the filled quantity.

Feedback from the project The incentive is not required to be proportional to the fill amount. The incentive is collected based on the actual fees generated by the taker, which is our business logic.

2.1.9 Inconsistent `exchange` validation in function `updateStockInfo()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `updateStockInfo()` is a privileged function that allows direct updates to a stock token's `companyName` and `exchange`. However, the protocol maintains a registry of valid `exchange` in the `StockTokenFactory` contract via the `exchangeNames` mapping, where each `exchange ID` corresponds to a predefined `exchange`.

The function `updateStockInfo()` accepts an `exchange` string directly and does not validate whether the provided value exists in `StockTokenFactory.exchangeNames`. This lack of validation may result in stock tokens being assigned exchange values that are not registered or recognized by the factory contract, leading to inconsistent exchange metadata across the protocol.

```
function updateStockInfo(address token, string calldata companyName, string calldata exchange)
    external
    onlyMinter
    whenNotPaused
{
    if (!isRegisteredToken[token]) revert TokenNotRegistered();
    IStockToken(token).updateStockInfo(companyName, exchange);
}
```

Listing 2.16: src/StockTokenManager.sol

Impact Stock tokens may be assigned `exchange` values that are not registered in `StockTokenFactory`, leading to inconsistent or invalid exchange metadata.

Suggestion Add a validation to ensure that the provided `exchange` is registered in the contract `StockTokenFactory`.

2.1.10 Inconsistent lock window configuration

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `RFQSettlement` contract sets `DEFAULT_LOCK_WINDOW` to 360 days and `MAX_LOCK_WINDOW` to 720 days, while the initialization comments indicate an intended lock duration of one minute. This mismatch creates inconsistency between the configured lock duration and the documented expectation, potentially causing orders to remain locked far longer than intended.

```
uint256 private constant DEFAULT_LOCK_WINDOW = 360 * 24 * 60 * 60; // 360 days default lock window
uint256 private constant MAX_LOCK_WINDOW = 720 * 24 * 60 * 60; // 720 days maximum lock window
```

Listing 2.17: src/RFQSettlement.sol

```
    lockWindow = DEFAULT_LOCK_WINDOW; // Default 60 seconds (1 minute) lock window
```

Listing 2.18: src/RFQSettlement.sol

Impact User funds may be locked for extended periods, limiting accessibility and order execution.

Suggestion Update the lock window constants to match the intended duration documented in the initialization comments.

2.2 Recommendation

2.2.1 Improper logic in function `computeAddress()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `computeAddress()` is a view function intended to deterministically compute a `stockToken` address given certain parameters. However, its inputs do not include `companyName`. In function `createStockToken()`, `companyName` is part of the initialization parameters, and the `CREATE2` derived address is determined by the `deployer`, `salt`, and the hash of the `init_code` (which includes constructor/initialization arguments). As a result, function `computeAddress()` may compute an address that differs from the actual address deployed by function `createStockToken()`.

```
function computeAddress(string calldata ticker, uint16 exchange, uint256 nonce) external view returns (address) {
        bytes32 salt = keccak256(abi.encodePacked(ticker, exchange, nonce));
        string memory exchangeName = exchangeNames[exchange];
        bytes32 bytecodeHash = keccak256(
            abi.encodePacked(
                type(StockToken).creationCode, abi.encode(ticker, "", exchangeName, address(this),
                    manager)
            )
        );

        return address(uint160(uint256(keccak256(abi.encodePacked(bytes1(0xff), address(this), salt
            , bytecodeHash)))));
    }
```

Listing 2.19: src/StockTokenFactory.sol

Suggestion Include `companyName` as an input parameter to function `computeAddress()`.

2.2.2 Inconsistent event emission in `invalidateNonce()` and `invalidateNonces()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `Permit2`, the functions `invalidateNonce()` and `invalidateNonces()` both emit the `NonceInvalidated` event, but the semantics of the emitted values are inconsistent. The `invalidateNonce()` function emits the specific nonce that has been invalidated. The `invalidateNonces()` function emits the new `currentNonce`, implying that all nonces below this value are invalidated. This inconsistency creates ambiguity in the conveyed meaning.

```
    /**
     * @notice Invalidate all nonces up to current nonce
     * @dev Increments currentNonce, invalidating all previous nonces
     */
    function invalidateNonces() external {
        currentNonce[msg.sender]++;
        emit NonceInvalidated(msg.sender, currentNonce[msg.sender]);
    }
```

Listing 2.20: src/Permit2.sol

```
    function invalidateNonce(uint256 nonce) external {
        nonces[msg.sender][nonce] = true;
        emit NonceInvalidated(msg.sender, nonce);
    }
```

Listing 2.21: src/Permit2.sol

Suggestion Revise the logic to ensure that the emitted value accurately reflects either the specific invalidated nonce or the updated `currentNonce`.

2.3 Note

2.3.1 Potential centralization risks

Introduced by Version 1

Description The protocol relies on several privileged roles, including the owner and manager, to perform sensitive operations, introducing potential centralization risks. The owner has the authority to pause or unpause the protocol, configure global parameters, grant privileged roles such as Minter and Operator, and transfer ownership directly through function transferOwnership(). The protocol is deployed behind a UUPSUpgradeable proxy, and contract upgrades are controlled via the proxy with restricted access, typically limited to the owner or designated administrators. If the private keys of these privileged accounts are lost or maliciously exploited, the protocol could be significantly compromised.

2.3.2 Trusted off-chain logic

Introduced by Version 1

Description The protocol depends on off-chain decision-making for critical operations. Takers rely on off-chain signals to identify eligible orders and claim incentives for executing them on-chain. Similarly, corporate actions such as stock dividends, stock splits, and reverse splits are determined off-chain, including all related parameters like dividend taxes, and are directly executed on-chain by the operator based on these externally provided values. The protocol does not independently verify or compute these outcomes, making it fully dependent on the accuracy and integrity of the off-chain decisions.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS