CODESPECT

Carina Finance contracts

SECURITY ASSESSMENT REPORT

October 17, 2025

Prepared for:

Carina

Contents

1 About CODESPECT

CODESPECT is a specialized smart contract security firm dedicated to ensure the safety, reliability, and success of blockchain projects. Our services include comprehensive smart contract audits, secure design and architecture consultancy, and smart contract development across leading blockchain platforms such as Ethereum (Solidity), Starknet (Cairo), and Solana (Rust).

At CODESPECT, we are committed to build secure, resilient blockchain infrastructures. We provide strategic guidance and technical expertise, working closely with our partners from concept development through deployment. Our team consists of blockchain security experts and seasoned engineers who apply the latest auditing and security methodologies to help prevent exploits and vulnerabilities in your smart contracts.

Smart Contract Auditing: Security is at the core of everything we do at CODESPECT. Our auditors conduct thorough security assessments of smart contracts written in Solidity, Cairo, and Rust, ensuring that they function as intended without vulnerabilities. We specialize in providing tailored security solutions for projects on EVM-compatible chains and Starknet. Our audit process is highly collaborative, keeping clients involved every step of the way to ensure transparency and security. Our team is also dedicated to cutting-edge research, ensuring that we stay ahead of emerging threats.

Secure Design & Architecture Consultancy: At CODESPECT, we believe that secure development begins at the design phase. Our consultancy services offer deep insights into secure smart contract architecture and blockchain system design, helping you build robust, secure, and scalable decentralized applications. Whether you're working with Ethereum, Starknet, or other blockchain platforms, our team helps you navigate the complexity of blockchain development with confidence.

Tailored Cybersecurity Solutions: CODESPECT offers specialized cybersecurity solutions designed to minimize risks associated with traditional attack vectors, such as phishing, social engineering, and Web2 vulnerabilities. Our solutions are crafted to address the unique security needs of blockchain-based applications, reducing exposure to attacks and ensuring that all aspects of the system are fortified.

With a focus on the intersection of security and innovation, CODESPECT strives to be a trusted partner for blockchain projects at every stage of development and for each aspect of security.

2 Disclaimer

Limitations of this Audit: This report is based solely on the materials and documentation provided to CODESPECT for the specific purpose of conducting the security review outlined in the Summary of Audit and Files. The findings presented in this report may not be comprehensive and may not identify all possible vulnerabilities. CODESPECT provides this review and report on an "as-is" and "as-available" basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, is entirely at your own risk.

Inherent Risks of Blockchain Technology: Blockchain technology is still evolving and is inherently subject to unknown risks and vulnerabilities. This review focuses exclusively on the smart contract code provided and does not cover the compiler layer, underlying programming language elements beyond the reviewed code, or any other potential security risks that may exist outside of the code itself.

Purpose and Reliance of this Report: This report should not be viewed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. Third parties should not rely on this report for any purpose, including making decisions related to investments or purchases.

Liability Disclaimer: To the maximum extent permitted by law, CODESPECT disclaims all liability for the contents of this report and any related services or products that arise from your use of it. This includes but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Third-Party Products and Services: CODESPECT does not warrant, endorse, or assume responsibility for any third-party products or services mentioned in this report, including any open-source or third-party software, code, libraries, materials, or information that may be linked to, referenced by, or accessible through this report. CODESPECT is not responsible for monitoring any transactions between you and third-party providers. We strongly recommend conducting thorough due diligence and exercising caution when engaging with third-party products or services, just as you would for any other product or service transaction.

Further Recommendations: We advise clients to schedule a re-audit after any significant changes to the codebase to ensure ongoing security and reduce the risk of newly introduced vulnerabilities. Additionally, we recommend implementing a bug bounty program to incentivize external developers and security researchers to identify and disclose potential vulnerabilities safely and responsibly.

Disclaimer of Advice: FOR AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, AND ANY ASSOCIATED SERVICES OR MATERIALS SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

3 Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Table 1: Risk Classification Matrix based on Likelihood and Impact

3.1 Impact

- **High** - Results in a substantial loss of assets (more than 10%) within the protocol or causes significant disruption to the majority of users.
- **Medium** - Losses affect less than 10% globally or impact only a portion of users, but are still considered unacceptable.
- **Low** - Losses may be inconvenient but are manageable, typically involving issues like griefing attacks that can be easily resolved or minor inefficiencies such as gas costs.

3.2 Likelihood

- **High** - Very likely to occur, either easy to exploit or difficult but highly incentivized.
- **Medium** - Likely only under certain conditions or moderately incentivized.
- **Low** - Unlikely unless specific conditions are met, or there is little-to-no incentive for exploitation.

3.3 Action Required for Severity Levels

- **Critical** - Must be addressed immediately if already deployed.
- **High** - Must be resolved before deployment (or urgently if already deployed).
- **Medium** - It is recommended to fix.
- **Low** - Can be fixed if desired but is not crucial.

In addition to High, Medium, and Low severity levels, CODESPECT utilizes two other categories for findings: **Informational** and **Best Practices**.

a) **Informational** findings do not pose a direct security risk but provide useful information the audit team wants to communicate formally.

b) **Best Practices** findings indicate that certain portions of the code deviate from established smart contract development standards.

4 Executive Summary

This document presents the security assessment conducted by CODESPECT for the smart contracts of Carina Finance. Carina is an intent-based decentralised exchange (DEX) aggregator built on the SEI blockchain.

This audit focused on the settlement contracts of the aggregator, including all associated libraries and components responsible for the creation and management of native token orders.

The audit was performed using:

a) Manual analysis of the codebase.

b) Dynamic analysis of smart contracts, execution testing.

CODESPECT found two points of attention, one classified as `Medium` and one classified as `Informational`. All of the issues are summarised in Table 2.

Organisation of the document is as follows:

- **Section 5** summarizes the audit.
- **Section 6** describes the functionality of the code in scope.
- **Section 7** presents the issues.
- **Section 8** presents the additional notes of the auditors.
- **Section 9** discusses the documentation provided by the client for this audit.
- **Section 10** presents the compilation and tests.

Issues found:

Severity	Unresolved	Fixed	Acknowledged
Medium	0	1	0
Informational	0	0	1
Total	**0**	**1**	**1**

Table 2: Summary of Unresolved, Fixed, and Acknowledged Issues

5 Audit Summary

Audit Type	Security Review
Project Name	Carina Finance
Type of Project	Intent based DEX aggregator
Duration of Engagement	4 Days
Duration of Fix Review Phase	1 Day
Draft Report	October 17, 2025
Final Report	October 17, 2025
Repository	carina-sc
Commit (Audit)	6ff2ad112b6b62acb054411b8181d9414970b08b
Commit (Final)	3adbb8b541e3be631235179acfe1b3a416b2c230
Documentation Assessment	High
Test Suite Assessment	High
Auditors	Talfao, suspiciousbandicoot

Table 3: Summary of the Audit

5.1 Scope - Audited Files

	Contract	**LoC**
1	Settlement.sol	157
2	NativeTokenFlow.sol	124
3	VaultRelayer.sol	22
4	Authenticator.sol	17
5	mixins/OrderSigning.sol	138
6	interfaces/ISettlement.sol	62
7	interfaces/INativeTokenFlow.sol	39
8	interfaces/IAuthenticator.sol	6
9	interfaces/IVaultRelayer.sol	6
10	interfaces/IEIP1271.sol	5
11	interfaces/IWrappedNativeToken.sol	6
12	libraries/TransferHelper.sol	68
13	libraries/OrderHelper.sol	44
14	libraries/TradeHelper.sol	33
15	libraries/CallDataHelper.sol	10
	Total	**737**

5.2 Findings Overview

	Finding	**Severity**	**Update**
1	Malicious solver can drain `NativeTokenFlow` contract via reentrancy	Medium	Fixed
2	EIP-1271 signature verification allows for a callback to user-controlled contracts	Info	Acknowledged

6 System Overview

Carina Finance is a decentralized exchange (DEX) aggregator built on the **SEI blockchain**. The reviewed scope covers the settlement layer of Carina, which serves as the core mechanism through which solvers execute and settle users' signed orders to perform trades on their behalf.

The main components of the protocol are:

- `Authenticator` — Manages the list of active solvers within the Carina protocol.
- `VaultRelayer` — A contract approved by users to transfer their `tokenIn` assets. The `Settlement` contract utilises this relayer to execute the actual token transfers required for trade settlement.
- `NativeTokenFlow` — Facilitates the creation of native token orders. Users deposit native SEI tokens into this contract, which are later wrapped by the solver during settlement and used through the `Settlement` contract.
- `Settlement` — The central contract responsible for verifying signed orders, executing solver actions, and ensuring that users receive the correct `tokenOut` amount as defined in their signed orders.

6.1 Common Trade Flow

The process for a standard ERC-20 trade begins when a user requests a swap via the Carina Finance front-end. The platform's off-chain service generates a quote detailing the terms of the trade, including a protocol fee. The user signs this quote, creating a cryptographically secure order. This signed order is then made available to a network of Solvers, who compete to execute it on-chain.

The `Order` structure is defined as follows:

```
struct Order {
    address receiver;
    uint64 validTo;
    IERC20 tokenIn;
    IERC20 tokenOut;
    uint256 amountIn;
    uint256 amountOut;
    uint256 minAmountOut;
    uint256 feeAmount;
}
```

A user specifies the receiver, the expiration timestamp, input and output tokens, and most importantly, the `minAmountOut` parameter — which defines the minimum number of tokens the user expects to receive. This serves as a primary protection mechanism against any malicious solver behavior.

During settlement, the solver calls the `settle(...)` function:

```
function settle(
    IERC20[] calldata tokens,
    uint256[] calldata clearingPrices,
    Trade[] calldata trades,
    Action[][3] calldata actions
) external;
```

The solver inputs all trades to be executed (from which user orders are recovered and verified, along with the signing scheme), clearing prices, and actions. The solver is responsible for correctly setting all token prices and handling decimal precision differences. However, users remain protected by their `minAmountOut` value — if not satisfied, the transaction reverts automatically.

The solver also provides three batches of arbitrary actions to be executed from the `Settlement` contract, enabling flexible trade composition.

Once a trade is successfully executed, the user receives the expected `tokenOut` amount, and the protocol collects the specified `feeAmount` from the quote.

Users can also cancel a signed order before it is settled by calling the `invalidateOrder(...)` function, which ensures that any subsequent settlement attempt with this order reverts:

```
function invalidateOrder(bytes calldata orderId) external;
```

6.2 Native Token Trade Flow

The native token trade flow functions similarly to the common trade flow, with the main difference being that the user must create the order through the `NativeTokenFlow` contract. In this case, the user sends their native SEI tokens directly to the contract. The user can also cancel the order at any time prior to settlement, which results in a refund of the unfilled portion of the order.

Another key difference in this flow is the verification mechanism during settlement. When the solver settles a native token order, the `Settlement` contract utilizes the **EIP-1271** standard to verify the validity of the order's signature. Specifically, the `Settlement` contract calls the `isValidSignature(...)` function on the `NativeTokenFlow` contract, which returns the expected magic value based on the order's status and expiration timestamp.

```
function isValidSignature(
    bytes32 _hash,
    bytes memory _signature
) external view returns (bytes4 magicValue);
```

This mechanism ensures that native SEI orders are validated in compliance with the EIP-1271 standard, allowing for secure and verifiable off-chain signatures during the settlement process.

7 Issues

7.1 [Medium] Malicious solver can drain `NativeTokenFlow` contract via reentrancy

File(s): `NativeTokenFlow.sol`

Description: The `NativeTokenFlow` contract allows users to create an order using the native token. Such an order is then fulfilled by a solver through the `Settlement` contract, after which the user receives the desired assets. A user can cancel their order via `cancelOrder(...)` before it is settled. If the order is successfully cancelled, the user is refunded the corresponding native token amount.

```
if (orderFilledAmount < order.amountIn) {
    uint256 refundAmount = order.amountIn - orderFilledAmount;
    if (refundAmount > 0) {
        // ...

        (bool success,) = payable(orderMeta.owner).call{value: refundAmount}("");
        if (!success) {
            revert NativeTokenTransferFailed();
        }
    }
}
// ...
}
// ...
orderMeta.status = NATIVE_FLOW_ORDER_STATUS_CANCELLED;
```

However, as shown above, the cancellation status for the order is set **after** the low-level `call(...)`, which introduces a **reentrancy vector**.

A malicious solver could exploit this vulnerability as follows:

a. Create their own order in the `NativeTokenFlow` contract;
b. Call `cancelOrder(...)` in the same contract;
c. When the refund is issued via `call(...)`, the attacker's `receive(...)` function is triggered. From there, they can call `settle(...)` on the `Settlement` contract. This succeeds because `NativeTokenFlow::isValidSignature(...)` still considers the order valid (its status remains `NATIVE_FLOW_ORDER_STATUS_CREATED`).
d. As a result, the solver effectively uses the same `tokenIn` amount twice — first receiving a refund, and then fulfilling the order again.

Impact: Potential draining of the `NativeTokenFlow` contract. However, the overall impact is constrained by the actor's ability to perform the attack and the available liquidity in the contract.

Recommendation(s): Set the cancellation status **before** performing the low-level call, e.g.:

```
orderMeta.status = NATIVE_FLOW_ORDER_STATUS_CANCELLED;

// ...
(bool success,) = payable(orderMeta.owner).call{value: refundAmount}("");
```

Status: Fixed

Update from Carina: Fixed in 3adbb8b541e3be631235179acfe1b3a416b2c230

7.2 [Info] EIP-1271 signature verification allows for a callback to user-controlled contracts

File(s): `mixins/OrderSigning.sol`

Description: The settlement process, initiated by a Solver's call to the `settle(...)` function, involves verifying user signatures for each trade. The protocol supports various signing schemes, including EIP-1271 for smart contract-based wallets. The verification for this scheme is handled in the `recoverEIP1271Signer(...)` function.

This function makes an external call to the user's contract to invoke `isValidSignature(...)`, as required by the EIP-1271 standard. This callback occurs after the pre-settlement actions (`actions[0]`) have been executed but before the main liquidity-providing actions (`actions[1]`) and the final token transfers.

This creates an attack surface where a malicious user, via their smart contract wallet, can execute arbitrary logic during the settlement process. This could be used to manipulate the state of external protocols (e.g., AMM pool prices) that the Solver's actions rely on. Such manipulation could lead to outcomes like transaction reverts due to slippage (denial of service against the Solver) or achieving a more favorable trade execution for the user at the Solver's expense.

```
// src/mixins/OrderSigning.sol

function recoverEip1271Signer(bytes32 orderDigest, bytes calldata encodedSignature)
    internal
    view
    returns (address owner)
{
    assembly {
        // owner = address(encodedSignature[0:20])
        owner := shr(96, calldataload(encodedSignature.offset))
    }

    bytes calldata signature = encodedSignature[20:];

    // @audit-issue This makes an arbitrary external call to a user-controlled contract.
    // It executes between the pre-settlement (setup) and main settlement actions.
    if (IEIP1271(owner).isValidSignature(orderDigest, signature) != EIP1271_MAGIC_VALUE) {
        revert InvalidEip1271Signature();
    }
}
```

Unlike traditional front-running, the logic that affects the Solver is encoded into the order's own verification process via the smart contract wallet. This means that private mempools and other front-running protections are ineffective against this vector.

Impact: This issue is informational, as the behaviour is part of a standard integration. It aims to highlight the potential risk for Solvers, who are the primary actors exposed to this manipulation vector.

Recommendation(s): Consider documenting this behaviour to ensure that Solvers are aware of the potential for re-entrancy through EIP-1271 signature verification. Solvers should implement their own safeguards, such as performing pre-flight checks on external conditions and setting strict slippage parameters within their settlement actions to mitigate manipulation risk.

Status: Acknowledged

Update from Carina: We acknowledge it and will document it in the solver integration guide for Carina.

8 Additional Notes

This section provides supplementary auditor observations regarding the code. These points were not identified as individual issues but serve as informative recommendations to enhance the overall quality and maintainability of the codebase.

- The solver is responsible for providing the correct decimal precision for prices. Since different tokens may use varying numbers of decimals, it is up to the solver to ensure accurate price scaling to correctly calculate the `amountOut` for the target token.
- The protocol's design entrusts solvers with providing accurate `clearingPrices` for token swaps. There is no on-chain validation to ensure these prices align with current market rates. While users are protected by the signed `minAmountOut` parameter, this model relies on an off-chain auction and penalty system to incentivize solvers to offer fair execution and prevent them from consistently providing prices that only meet the minimum required output.

9 Evaluation of Provided Documentation

The **Carina Finance** documentation was provided as a comprehensive code walkthrough, where all functionalities and flows were explained. Additionally, the code included NatSpec comments detailing the intended purpose of each function and the reasoning behind most of the implemented logic. A unified document was also provided as audit-supporting material, outlining the high-level design and the inspiration behind the protocol's contracts.

The documentation could be further enhanced by:

- **Technical documentation** accessible via Carina's website, explaining the overall functionality and the integration with the solver.

Overall, the documentation provides a thorough description of the protocol. Furthermore, the Carina team was consistently available to address any questions or concerns raised by the CODESPECT team.

10 Test Suite Evaluation

10.1 Compilation Output

```
> forge build

[] Compiling...
[] Compiling 19 files with Solc 0.8.29
[] Solc 0.8.29 finished in 21.73s
Compiler run successful!
...
```

10.2 Tests Output

```
> forge test

Ran 7 tests for test/libraries/OrderHelperTest.t.sol:OrderHelperTest
[PASS] testOrderTypeHash() (gas: 309)
[PASS] testPackOrderIdInvalidLengthLong() (gas: 9638)
[PASS] testPackOrderIdInvalidLengthShort() (gas: 9550)
[PASS] testPackOrderIdValid() (gas: 4799)
[PASS] testUnpackOrderIdInvalidLengthLong() (gas: 9483)
[PASS] testUnpackOrderIdInvalidLengthShort() (gas: 9590)
[PASS] testUnpackOrderIdValid() (gas: 6887)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 5.85ms (6.70ms CPU time)

Ran 2 tests for test/VaultRelayerTest.t.sol:VaultRelayerTest
[PASS] testTransferFromAccounts_RevertIfNotOwner() (gas: 15670)
[PASS] testTransferFromAccounts_Success() (gas: 450563)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 13.80ms (6.00ms CPU time)

Ran 2 tests for test/AuthenticatorTest.t.sol:AuthenticatorTest
[PASS] testSetSolverStatus_RevertIfNotOwner() (gas: 17845)
[PASS] testSetSolverStatus_Success() (gas: 85338)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 13.96ms (3.08ms CPU time)

Ran 5 tests for test/libraries/TransferHelperTest.sol:TransferHelperTest
[PASS] testTransferFromAccounts_RevertIfERC20TransferFails() (gas: 16382)
[PASS] testTransferFromAccounts_Success() (gas: 449227)
[PASS] testTransferToAccounts_RevertIfERC20TransferFails() (gas: 61970)
[PASS] testTransferToAccounts_RevertIfETHTransferFails() (gas: 72842)
[PASS] testTransferToAccounts_Success() (gas: 322567)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 13.92ms (7.18ms CPU time)

Ran 33 tests for test/mixins/OrderSigningTest.sol:OrderSigningTest
[PASS] testAllocateRecoveredOrder() (gas: 10824)
[PASS] testDomainSeparator() (gas: 6803)
[PASS] testEmptySignature() (gas: 17239)
[PASS] testInvalidTokenIndices() (gas: 60105)
[PASS] testMaximumValues() (gas: 35304)
[PASS] testMultiplePreSignatures() (gas: 72747)
[PASS] testOrderIdUniqueness() (gas: 174225)
[PASS] testRecoverEip1271SignerInvalidEncoding() (gas: 9880)
[PASS] testRecoverEip1271SignerInvalidSignature() (gas: 19560)
[PASS] testRecoverEip1271SignerNonContractAddress() (gas: 13917)
[PASS] testRecoverEip1271SignerSuccess() (gas: 36597)
[PASS] testRecoverEip712SignerInvalidSignature() (gas: 10514)
[PASS] testRecoverEip712SignerSuccess() (gas: 35741)
[PASS] testRecoverEip712SignerWrongSigner() (gas: 33264)
[PASS] testRecoverEthsignSignerInvalidSignature() (gas: 10313)
[PASS] testRecoverEthsignSignerSuccess() (gas: 36048)
[PASS] testRecoverEthsignSignerWrongDigest() (gas: 33664)
[PASS] testRecoverOrderFromTradeEip1271() (gas: 172220)
[PASS] testRecoverOrderFromTradeEip712() (gas: 154314)
[PASS] testRecoverOrderFromTradeEthSign() (gas: 174057)
```

```
[PASS] testRecoverOrderFromTradePreSign() (gas: 130063)
[PASS] testRecoverOrderFromTradeWithExplicitReceiver() (gas: 158230)
[PASS] testRecoverPermit2WitnessSignerInvalidPermit2WitnessSignature() (gas: 16494)
[PASS] testRecoverPermit2WitnessSignerSuccess() (gas: 14459)
[PASS] testRecoverPreSignerInvalidSignatureLength() (gas: 10638)
[PASS] testRecoverPreSignerNotPreSigned() (gas: 14741)
[PASS] testRecoverPreSignerRevokedSignature() (gas: 31927)
[PASS] testRecoverPreSignerSuccess() (gas: 39516)
[PASS] testSetPreSignedFailsForNonOwner() (gas: 13411)
[PASS] testSetPreSignedInvalidOrderId() (gas: 12059)
[PASS] testSetPreSignedRevoke() (gas: 35453)
[PASS] testSetPreSignedSuccess() (gas: 42716)
[PASS] testZeroAddressHandling() (gas: 154157)
Suite result: ok. 33 passed; 0 failed; 0 skipped; finished in 14.79ms (41.26ms CPU time).

Ran 26 tests for test/NativeTokenFlowTest.t.sol:NativeTokenFlowTest
[PASS] testCancelOrder_RevertIfNotOrderOwnerAndOrderNotExpired() (gas: 74234)
[PASS] testCancelOrder_RevertIfOrderAlreadyCancelled() (gas: 115822)
[PASS] testCancelOrder_RevertIfOrderDoesNotExist() (gas: 36755)
[PASS] testCancelOrder_RevertIfOrderSettled() (gas: 334245)
[PASS] testCancelOrder_RevertIfOrderSettledAndExceededValidTo() (gas: 336160)
[PASS] testCancelOrder_RevertWhenNativeTokenTransferFails() (gas: 75020)
[PASS] testCancelOrder_SuccessIfContractHadSomeWrappedNativeToken() (gas: 141518)
[PASS] testCancelOrder_SuccessIfNotOrderOwnerAndOrderExpired() (gas: 118941)
[PASS] testCancelOrder_SuccessIfOrderExpired() (gas: 116700)
[PASS] testCancelOrder_SuccessIfOrderNotExpired() (gas: 116990)
[PASS] testCancelOrdersIgnoreInvalidCancellation_IgnoreSettledOrdersAndExceedValidTo() (gas: 411767)
[PASS] testCancelOrdersIgnoreInvalidCancellation_Success() (gas: 158263)
[PASS] testCancelOrdersIgnoreInvalidCancellation_SuccessIfSomeOrdersInvalid() (gas: 446375)
[PASS] testCreateOrder_RevertIfInvalidValidTo() (gas: 24484)
[PASS] testCreateOrder_RevertIfNotEnoughNativeToken() (gas: 24346)
[PASS] testCreateOrder_RevertIfOrderAlreadyExists() (gas: 73941)
[PASS] testCreateOrder_RevertIfReceiverIsZeroAddress() (gas: 24693)
[PASS] testCreateOrder_RevertIfTooMuchNativeToken() (gas: 24236)
[PASS] testCreateOrder_RevertIfZeroAmountIn() (gas: 24325)
[PASS] testCreateOrder_Success() (gas: 335180)
[PASS] testCreateOrder_SuccessWithFee() (gas: 360457)
[PASS] testIsValidSignature() (gas: 116593)
[PASS] testSettlement_RevertIfOrderCancelled() (gas: 167667)
[PASS] testSettlement_RevertIfOrderNotExist() (gas: 84198)
[PASS] testWrapAllNativeToken_Success() (gas: 79287)
[PASS] testWrapNativeToken_Success() (gas: 79502)
Suite result: ok. 26 passed; 0 failed; 0 skipped; finished in 15.64ms (13.28ms CPU time)

Ran 23 tests for test/SettlementTest.t.sol:SettlementTest
[PASS] testInvalidateOrder_RevertIfNotOwner() (gas: 23571)
[PASS] testInvalidateOrder_RevertIfOrderAlreadyFilled() (gas: 235841)
[PASS] testInvalidateOrder_RevertIfOrderAlreadyInvalidated() (gas: 51612)
[PASS] testInvalidateOrder_Success() (gas: 50798)
[PASS] testSetFeeReceiver_RevertIfNotOwner() (gas: 13236)
[PASS] testSetFeeReceiver_RevertIfZeroAddress() (gas: 14049)
[PASS] testSetFeeReceiver_Success() (gas: 28217)
[PASS] testSetPreSigned_RevertIfNotOwner() (gas: 9697)
[PASS] testSetPreSigned_Success() (gas: 30446)
[PASS] testRecoverOrderFromTradePreSign() (gas: 130063)
[PASS] testRecoverOrderFromTradeWithExplicitReceiver() (gas: 158230)
[PASS] testRecoverPermit2WitnessSignerInvalidPermit2WitnessSignature() (gas: 16494)
[PASS] testRecoverPermit2WitnessSignerSuccess() (gas: 14459)
[PASS] testRecoverPreSignerInvalidSignatureLength() (gas: 10638)
[PASS] testRecoverPreSignerNotPreSigned() (gas: 14741)
[PASS] testRecoverPreSignerRevokedSignature() (gas: 31927)
[PASS] testRecoverPreSignerSuccess() (gas: 39516)
[PASS] testSetPreSignedFailsForNonOwner() (gas: 13411)
[PASS] testSetPreSignedInvalidOrderId() (gas: 12059)
[PASS] testSetPreSignedRevoke() (gas: 35453)
[PASS] testSetPreSignedSuccess() (gas: 42716)
[PASS] testZeroAddressHandling() (gas: 154157)
Suite result: ok. 33 passed; 0 failed; 0 skipped; finished in 14.79ms (41.26ms CPU time).
```

```
[PASS] testSettle_ExecuteActionsSuccess() (gas: 386756)
[PASS] testSettle_RevertIfActionReverts() (gas: 72273)
[PASS] testSettle_RevertIfCallActionToVaultRelayer() (gas: 98732)
[PASS] testSettle_RevertIfFeeGreaterThanOrEqualToAmountIn() (gas: 117382)
[PASS] testSettle_RevertIfNotEnoughAmountOut() (gas: 86822)
[PASS] testSettle_RevertIfNotSolver() (gas: 30384)
[PASS] testSettle_RevertIfOrderExpired() (gas: 86064)
[PASS] testSettle_RevertIfOrderIsInvalided() (gas: 264669)
[PASS] testSettle_RevertIfOrderIsSettledTwice() (gas: 323403)
[PASS] testSettle_RevertIfSigningSchemaEIP712Invalid() (gas: 82192)
[PASS] testSettle_RoundingFloorAmountOut() (gas: 309593)
[PASS] testSettle_SuccessIfSigningSchemaEIP712() (gas: 553679)
[PASS] testSettle_SuccessWithFeeAndERC20Transfer() (gas: 340824)
[PASS] testSettle_SuccessWithFeeAndPermit2Transfer() (gas: 383270)
Suite result: ok. 23 passed; 0 failed; 0 skipped; finished in 15.60ms (27.00ms CPU time)

Ran 4 tests for test/libraries/TradeHelperTest.t.sol:TradeHelperTest
[PASS] testExtractFlags() (gas: 12251)
[PASS] testExtractOrderValid() (gas: 26760)
[PASS] testFuzzExtractFlags(uint8,uint8,uint240) (runs: 256, : 5930, ~: 6120)
[PASS] testFuzzExtractOrder(address,uint32,uint32,uint32,uint256,uint256,uint256,uint256,uint8,uint8) (runs: 256, :
 ↪  15158, ~: 15263)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 28.19ms (38.64ms CPU time)

Ran 8 test suites in 115.40ms (121.76ms CPU time): 102 tests passed, 0 failed, 0 skipped (102 total tests)
```

10.3 Notes on the Test Suite

The provided test suite demonstrates exceptional coverage across all core functionalities, including the integrated libraries.

- It achieves **100% coverage**, indicating a highly comprehensive and well-maintained testing framework.
- **Fuzz testing** has been incorporated for selected functionalities, enhancing robustness against edge cases and unexpected inputs.
- The suite also includes **invariant tests**, ensuring that key protocol properties consistently hold under various conditions.

Overall, the testing framework reflects a strong commitment to quality assurance and reliability.