MixBytes()

BarterDAO Superposition Security Audit Report

NOVEMBER 07, 2025

Table of Contents

1. Introduction

1.1 Disclaimer

The audit makes no statements or warranties regarding the utility, safety, or security of the code, the suitability of the business model, investment advice, endorsement of the platform or its products, the regulatory regime for the business model, or any other claims about the fitness of the contracts for a particular purpose or their bug-free status.

1.2 Executive Summary

Superposition is a decentralized order book protocol that allows users to create and fill orders for token swaps. The protocol consists of a SuperpositionVault that handles order execution and a SuperpositionRouter that provides a simplified interface for order filling with callback functionality.

In this audit, we focused on examining attack vectors related to signature validation, order execution logic, replay attacks, reentrancy vulnerabilities, callback mechanisms, and token transfer operations.

We also went through our detailed checklist, covering other aspects such as business logic, common ERC20 issues, interactions with external contracts, integer overflows, reentrancy attacks, access control, typecast pitfalls, rounding errors and other potential issues.

The audit identified vulnerabilities related to replay attack protection, reentrancy in callback mechanisms, and precision loss in order calculations.

Key notes and recommendations:

- **The audit assumes that SuperpositionRouter should not retain any funds or approvals after any transaction completion, as otherwise anyone could withdraw them later.**
- The protocol does not support fee-on-transfer tokens.
- In `swap()`: the maker may receive 1 wei less than expected due to gas optimization.
- Signers may be set to zeros, effectively pausing the protocol.
- Any single owner can reset signer addresses to zero without approval from the others. This creates a risk: if one owner is compromised, an attacker can repeatedly reset the signers, effectively putting the protocol on pause and requiring redeployment. All existing maker approvals would stay on the old contract, and users would need to grant new ones. Allowances in the old contract remain safe as long as no more than one owner is compromised.
- If two owners or all three order signers are compromised, the attacker can arbitrarily spend maker approvals at any price by signing orders with arbitrary data.
- For a public mempool, order execution is vulnerable to griefing attacks: since `filledTakerAmount` is not signed with the order, any caller can frontrun pending orders and execute them with minimal fill amounts. This attack vector is not relevant when transactions are routed through a private mempool.

1.3 Project Overview

Summary

Title	Description
Client Name	BarterLab
Project Name	Superposition
Type	Solidity
Platform	EVM
Timeline	18.09.2025-05.11.2025

Scope of Audit

File	Link
src/Errors.sol	Errors.sol
src/SuperpositionVault.sol	SuperpositionVault.sol
src/SuperpositionRouter.sol	SuperpositionRouter.sol
src/MinimalProxy.sol	MinimalProxy.sol

Versions Log

Date	Commit Hash	Note
18.09.2025	8dfd5bceb02bef6892a574da6862f673a5ea66b6	Initial Commit
03.11.2025	c3611a76302857f061ee74c860bdf872f8f933b2	Commit for re-audit
04.11.2025	700d53240381c6757d7374bf50e789b545c44494	Commit for re-audit

Mainnet Deployments

File	Address	Blockchain
SuperpositionVault.sol	0x8c649e9378fb6707b014714314d4e15ac297a6bf	Ethereum Mainnet
SuperpositionRouter.sol	0x0b7250866f0b014e6983cacc5b854eea7a3d9188	Ethereum Mainnet

1.4 Security Assessment Methodology

Project Flow

Stage	Scope of Work
Interim audit	**Project Architecture Review:** · Review project documentation · Conduct a general code review · Perform reverse engineering to analyze the project's architecture based solely on the source code · Develop an independent perspective on the project's architecture · Identify any logical flaws in the design OBJECTIVE: UNDERSTAND THE OVERALL STRUCTURE OF THE PROJECT AND IDENTIFY POTENTIAL SECURITY RISKS.
	Code Review with a Hacker Mindset: · Each team member independently conducts a manual code review, focusing on identifying unique vulnerabilities. · Perform collaborative audits (pair auditing) of the most complex code sections, supervised by the Team Lead. · Develop Proof-of-Concepts (PoCs) and conduct fuzzing tests using tools like Foundry, Hardhat, and BOA to uncover intricate logical flaws. · Review test cases and in-code comments to identify potential weaknesses. OBJECTIVE: IDENTIFY AND ELIMINATE THE MAJORITY OF VULNERABILITIES, INCLUDING THOSE UNIQUE TO THE INDUSTRY.
	Code Review with a Nerd Mindset: · Conduct a manual code review using an internally maintained checklist, regularly updated with insights from past hacks, research, and client audits. · Utilize static analysis tools (e.g., Slither, Mythril) and vulnerability databases (e.g., Solodit) to uncover potential undetected attack vectors. OBJECTIVE: ENSURE COMPREHENSIVE COVERAGE OF ALL KNOWN ATTACK VECTORS DURING THE REVIEW PROCESS.

Stage	Scope of Work
	Consolidation of Auditors' Reports: · Cross-check findings among auditors · Discuss identified issues · Issue an interim audit report for client review OBJECTIVE: COMBINE INTERIM REPORTS FROM ALL AUDITORS INTO A SINGLE COMPREHENSIVE DOCUMENT.
Re-audit	**Bug Fixing & Re-Audit:** · The client addresses the identified issues and provides feedback · Auditors verify the fixes and update their statuses with supporting evidence · A re-audit report is generated and shared with the client OBJECTIVE: VALIDATE THE FIXES AND REASSESS THE CODE TO ENSURE ALL VULNERABILITIES ARE RESOLVED AND NO NEW VULNERABILITIES ARE ADDED.
Final audit	**Final Code Verification & Public Audit Report:** · Verify the final code version against recommendations and their statuses · Check deployed contracts for correct initialization parameters · Confirm that the deployed code matches the audited version · Issue a public audit report, published on our official GitHub repository · Announce the successful audit on our official X account OBJECTIVE: PERFORM A FINAL REVIEW AND ISSUE A PUBLIC REPORT DOCUMENTING THE AUDIT.

1.5 Risk Classification

Severity Level Matrix

Severity	Impact: High	Impact: Medium	Impact: Low
Likehood: High	Critical	High	Medium
Likehood: Medium	High	Medium	Low
Likehood: Low	Medium	Low	Low

Impact

- **High** - Theft from 0.5% OR partial/full blocking of funds (>0.5%) on the contract without the possibility of withdrawal OR loss of user funds (>1%) who interacted with the protocol.
- **Medium** - Contract lock that can only be fixed through a contract upgrade OR one-time theft of rewards or an amount up to 0.5% of the protocol's TVL OR funds lock with the possibility of withdrawal by an admin.
- **Low** - One-time contract lock that can be fixed by the administrator without a contract upgrade.

Likelihood

- **High** - The event has a 50-60% probability of occurring within a year and can be triggered by any actor (e.g., due to a likely market condition that the actor cannot influence).
- **Medium** - An unlikely event (10-20% probability of occurring) that can be triggered by a trusted actor.
- **Low** - A highly unlikely event that can only be triggered by the owner.

Action Required

- **Critical** - Must be fixed as soon as possible.
- **High** - Strongly advised to be fixed to minimize potential risks.
- **Medium** - Recommended to be fixed to enhance security and stability.
- **Low** - Recommended to be fixed to improve overall robustness and effectiveness.

Finding Status

- **Fixed** - The recommended fixes have been implemented in the project code and no longer impact its security.
- **Partially Fixed** - The recommended fixes have been partially implemented, reducing the impact of the finding, but it has not been fully resolved.
- **Acknowledged** - The recommended fixes have not yet been implemented, and the finding remains unresolved or does not require code changes.

1.6 Summary of Findings

Findings Count

Severity	Count
Critical	1
High	1
Medium	3
Low	3

Findings Statuses

ID	Finding	Severity	Status
C-1	Double Order Attack via Callback Mechanism	Critical	Fixed
H-1	Replay Attack via Balance-Based Nonce	High	Fixed
M-1	Silent Truncation in Permit2 Transfers	Medium	Fixed
M-2	Malicious Signer Can Create Any Order and Spend Approvals	Medium	Fixed
M-3	DoS Attack via Dust Executions	Medium	Fixed
L-1	Approval Source Ambiguity	Low	Fixed
L-2	Precision Loss in Partial Swaps	Low	Fixed
L-3	Minor Issues	Low	Fixed

2. Findings Report

2.1 Critical

C-1	Double Order Attack via Callback Mechanism		
Severity	Critical	**Status**	Fixed in 700d5324

Description

The protocol transfers tokens from maker to taker first, and then calls back to the taker (i. e. msg.sender) to complete the order:

```
uint256 actualTakerAmount =
    filledtakerAmount < order.takerAmount ?
    filledtakerAmount : order.takerAmount;

uint256 balanceBefore =
    order.takerToken.balanceOf(address(order.maker));

ISuperpositionCallback(msg.sender).superpositionCallback(
        order, actualTakerAmount, callback);

uint256 balanceAfter =
    order.takerToken.balanceOf(address(order.maker));

// Check that callback provided enough tokens
if (balanceAfter < balanceBefore + actualTakerAmount) {
    revert ReceivedLessThanMinReturn(
        balanceAfter,
        balanceBefore + actualTakerAmount
    );
}
```

SuperpositionVault.sol#L141-L148
A malicious taker can exploit this by fulfilling another open order from the same maker during the callback (e.g., an order in the SuperPosition protocol, 1inch, etc.), causing the balance check to pass for both orders without actually transferring the required tokens in the first one.
Additionally, ERC-777 callbacks may be used for the attack.

Recommendation

We recommend calling safeTransferFrom() for both taker and maker directly in SuperpositionVault.swap() at the very end of the code and additionally protect it with a nonReentrant modifier as a precaution. For example:

```
contract SuperpositionVault {
    ...
    function swap(...) ... {
        ...
        ...
        ICallback(msg.sender).callback(...);
        ...
        ...

        token1.safeTransferFrom(maker, taker, ...);
        token2.safeTransferFrom(taker, maker, ...);

        emit ...;

        // no other code
    }
}
```

2.2 High

H-1	Replay Attack via Balance-Based Nonce		
Severity	High	**Status**	Fixed in c3611a76

Description

The protocol uses only the user's token balance as a nonce to prevent replay attacks:

```
uint256 currentMakerBalance = order.makerToken.balanceOf(order.maker);
if (currentMakerBalance < order.nonceBalance) {
    revert ReplayAttackDetected(currentMakerBalance, order.nonceBalance);
}
```

SuperpositionVault.sol#L115-L118

However, once the user's balance increases again (e.g., through receiving tokens from other sources), the same order can be executed multiple times if the deadline allows it. This creates a vulnerability where orders can be replayed whenever the maker's balance condition is satisfied again.

Recommendation

We recommend descending from the OpenZeppeling's Nonces contract:
Nonces.sol

2.3 Medium

M-1	Silent Truncation in Permit2 Transfers		
Severity	Medium	**Status**	Fixed in c3611a76

Description

The use of `uint160(amount)` in Permit2 transfers may silently truncate large amounts:

```
if (usePermit2) {
    permit2.transferFrom(
        order.maker,
        order.taker,
        uint160(amount),
        address(order.makerToken)
    );
} else {
    order.makerToken.safeTransferFrom(
        order.maker,
        order.taker,
        amount
    );
}
```

SuperpositionVault.sol#L134-L139

When casting a `uint256` amount to `uint160`, values larger than `2^160 - 1` will be silently truncated, potentially resulting in transferring significantly less than intended. For example, `1 << 161` **will truncate to zero**. However, this can be executed only if the transaction is signed by the signer.

Recommendation

We recommend checking `amount < type(uint160).max`.

M-2	Malicious Signer Can Create Any Order and Spend Approvals		
Severity	Medium	**Status**	Fixed in c3611a76

Description
In the current architecture, makers provide approvals to SuperpositionVault, which are then spent when executing orders that have valid signatures from the signer. The problem is that if the signer is compromised, they can create any order and spend any approval from the SuperpositionVault contract.
For example, suppose a maker has given approval to the contract to sell 1 BTC for 120,000 DAI. In this case, a compromised signer could sign and submit a different order - exchanging 1 BTC for 1 wei and specifying themselves as the taker, thus obtaining 1 BTC for free.
There have been numerous cases where motivated hackers have compromised central signers or even multiple signers, particularly in bridge protocols. Therefore, it is better to use a decentralized architecture where both maker and taker sign the transaction, so that orders cannot be arbitrarily created without the signature of the maker themselves.

Recommendation
We recommend checking signatures from both maker and taker instead of using a central signer.

Client's Commentary:
MixBytes: The implementation of three separate signature verification and deadline checks significantly mitigates the severity of this vulnerability. While the centralized signer architecture still poses a theoretical risk, requiring consensus from three independent signers combined with time-bound orders substantially reduces the attack surface and practical exploitability.

M-3	DoS Attack via Dust Executions		
Severity	Medium	**Status**	Fixed in c3611a76

Description
A griefer can frontrun any transaction with a signed order and execute it with an extremely small filledtakerAmount. Other users cannot complete the order because the nonceBalance check will fail:

```
uint256 currentMakerBalance = order.makerToken.balanceOf(order.maker);
if (currentMakerBalance < order.nonceBalance) {
    revert ReplayAttackDetected(currentMakerBalance, order.nonceBalance);
}
```

SuperpositionVault.sol#L125-L128

Recommendation
We recommend descending from Nonces.sol and implementing proper logic for multiple partial fills.

2.4 Low

L-1	Approval Source Ambiguity		
Severity	Low	**Status**	Fixed in c3611a76

Description
The protocol allows the executor to choose between Permit2 and direct token approval without this choice being part of the signed order:

```
if (usePermit2) {
    permit2.transferFrom(
        order.maker,
        order.taker,
        uint160(amount),
        address(order.makerToken)
    );
} else {
    order.makerToken.safeTransferFrom(
        order.maker,
        order.taker,
        amount
    );
}
```

SuperpositionVault.sol#L134-L139
This creates ambiguity about which approval source will be used. If a user provides unlimited approval to Permit2 and limited approval to the vault, expecting the limited approval to be used first, the use of Permit2 may be unexpected.

Recommendation
We recommend including the approval method (usePermit2 flag) in the order hash to ensure the maker explicitly specifies which approval mechanism should be used.

L-2	Precision Loss in Partial Swaps		
Severity	Low	**Status**	Fixed in c3611a76

Description

The protocol calculates the maker amount using division that can result in precision loss:

```
uint256 amount = filledtakerAmount < order.takerAmount
    ? (order.makerAmount * filledtakerAmount) / order.takerAmount
    : order.makerAmount;
```

SuperpositionVault.sol#L131-L134

For example, with makerAmount = 1.0 WBTC (1e8 wei), takerAmount = 120,000.00 DAI (120_000e18), and filledtakerAmount = 0.0011 DAI (i.e. takerAmount / makerAmount - 1), the calculated partial amount results in zero.

Recommendation

We recommend implementing minimum fill amount validation: ensure that filledtakerAmount is greater than several wei and also represents some minimum percentage of the total amount.

L-3	Minor Issues		
Severity	Low	**Status**	Fixed in c3611a76

Description

1. **Inconsistent token validation**: The check `if (order.makerToken == order.takerToken) revert InvalidSender();` should revert with a token-pair specific error.
2. **Duplicate deadline checks**: The codebase has both `<=` and `<` deadline checks - one should be standardized.

Recommendation

- Use appropriate error messages for token pair validation
- Standardize deadline comparison operators

3. About MixBytes

MixBytes is a leading provider of smart contract audit and research services, helping blockchain projects enhance security and reliability. Since its inception, MixBytes has been committed to safeguarding the Web3 ecosystem by delivering rigorous security assessments and cutting-edge research tailored to DeFi projects.

Our team comprises highly skilled engineers, security experts, and blockchain researchers with deep expertise in formal verification, smart contract auditing, and protocol research. With proven experience in Web3, MixBytes combines in-depth technical knowledge with a proactive security-first approach.

Why MixBytes

- **Proven Track Record:** Trusted by top-tier blockchain projects like Lido, Aave, Curve, and others, MixBytes has successfully audited and secured billions in digital assets.
- **Technical Expertise:** Our auditors and researchers hold advanced degrees in cryptography, cybersecurity, and distributed systems.
- **Innovative Research:** Our team actively contributes to blockchain security research, sharing knowledge with the community.

Our Services

- **Smart Contract Audits:** A meticulous security assessment of DeFi protocols to prevent vulnerabilities before deployment.
- **Blockchain Research:** In-depth technical research and security modeling for Web3 projects.
- **Custom Security Solutions:** Tailored security frameworks for complex decentralized applications and blockchain ecosystems.

MixBytes is dedicated to securing the future of blockchain technology by delivering unparalleled security expertise and research-driven solutions. Whether you are launching a DeFi protocol or developing an innovative dApp, we are your trusted security partner.

Contact Information

https://mixbytes.io/

https://github.com/mixbytes/audits_public

hello@mixbytes.io

https://x.com/mixbytes