0x
CrossChainReceiver

FINAL REPORT

March ‘2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Audit 0x - CrossChainReceiver
Website	0x.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/0xProject/0x-settler/commit/0ee37ec0a56ee5ded80280c76cdf3e5750821b7b
Resolution 1	https://github.com/0xProject/0x-settler/commit/30703307e85d16c094a1040b14f2ca89d369b502

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed Resolution
High					
Medium	2	2			
Low	1	1			
Informational	3			3	
Governance					
Total	6	3		3	

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium-level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless, the issue should be fixed immediately.
Informational	Effects are small and do not pose an immediate danger to the project or users.
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior.

3. Detection

MultiCallContext

The MultiCallContext contract is an abstract base contract that extends a standard Context contract to support advanced multicall and meta-transaction patterns, specifically for use with a trusted multicall contract at a fixed address.

- When a call is forwarded by the trusted multicall contract (MULTICALL_ADDRESS), the original sender's address is appended to the calldata. The contract's _msgSender and _msgData functions extract the true sender and strip the appended address from the calldata, so internal logic always sees the correct sender and data.
- The _isForwarded function detects if the current call is a forwarded call (either by the parent context or by the multicall contract).

Core Invariants:

INV 1: Only the contract at MULTICALL_ADDRESS is treated as a trusted forwarder.

INV 2: For direct calls, _msgSender returns the actual msg.sender.

INV 3: For direct calls, _msgData returns the full calldata

Privileged Functions

- none

Issue_01	Incorrect address alignment in _msgSender function leads to invalid sender address
Severity	**Medium**

Description	The _msgSender function attempts to extract the original sender from calldata when the call is forwarded by the trusted multicall contract (MULTICALL_ADDRESS). The relevant code is:
	
	sender :=
	    xor(
	        sender,
	        mul(
	            xor(calldataload(add(data.offset, sub(data.length, 0x14))), shl(0x60, sender)),
	            iszero(shl(0x60, xor(MULTICALL_ADDRESS, sender)))
	        )
	    )
	
	When sender == MULTICALL_ADDRESS, the appended address is loaded from calldata using calldataload. This loads the address left-aligned (in the upper 20 bytes of the 32-byte word), while the sender variable is right-aligned (in the lower 20 bytes). The outermost xor operation between these differently aligned addresses results in an incorrect address returned by _msgSender in the multicall case.
	
	Consequently, address checks that rely on _msgSender are affected. For example, the cleanup as well as the call function in the CrossChainReceiverFactory contract rely on it for an ownership check and are therefore subject to failure.

Recommendations	It is recommended to ensure matching/correct address alignment by applying appropriate bit-shifting. Furthermore, consider optimization by using eq(...) instead of iszero(xor(...)).
Comments / Resolution	Fixed, the address is now shifted correctly. An additional safety check was added in case the calldata is smaller or equal to 3 bytes as the multicall contract doesn't append the sender at the end of the calldata in this case.

	However, it needs to be noted that when the fallback function is triggered, the _msgSender() function will return the address of the Multicall contract rather than the original caller. While we do not recommend changing this implementation, it's important to be aware of this behavior, especially if _msgSender() were to be used within the fallback function. Consider documenting this behavior explicitly, as the Multicall contract behaves differently depending on the calldata size, which may not be immediately obvious to developers.

CrossChainReceiverFactory

The CrossChainReceiverFactory contract acts as a factory for deploying minimal proxy wallets at deterministic addresses using CREATE2. Its main role is to enable seamless, gas-efficient, and secure cross-chain swaps and fund reception for users interacting with bridges.

- The contract allows users to precompute the address of their wallet on any chain by using a Merkle root (of preapproved hashes) as the CREATE2 salt. This means funds can be sent to a wallet address before it is actually deployed.
- After deployment, the proxy can approve the Uniswap Permit2 contract to spend tokens. Permit2 manages nonce cancellation and stateless approvals, allowing the proxy to remain nearly stateless and secure.
- The contract supports ERC-7739 for advanced, nested, or fallback signature validation flows, enabling compatibility with smart contract wallets and complex authorization schemes.
- For workflows that cannot be fully supported gaslessly or with Permit2, the contract provides a generic call function, allowing the owner to execute arbitrary transactions from the proxy.

Core Invariants:

INV 1: The address of each minimal proxy is deterministically derived using CREATE2 with a salt based on the Merkle root and owner address.

INV 2: Only actions (e.g., swaps, calls) that are authorized by a valid Merkle proof (for a pre-approved hash) or a valid ERC-7739 signature are permitted.

INV 3: Only the current owner (as tracked by TwoStepOwnable) can execute arbitrary calls.

INV 4: The cleanup (selfdestruct) function can only be called by the owner or the factory itself, preventing unauthorized destruction.

Privileged Functions

- setOwner
- call
- cleanup

Issue_02	Frontrunning proxy deployment can cause loss of native funds
Severity	**Medium**
Description	When using the CrossChainReceiverFactory, users typically bridge native currency or ERC-20 tokens to a predetermined proxy address that will be deployed later. However, because anyone can deploy the proxy using the deploy function, a malicious actor can frontrun the legitimate deployment and implicitly call cleanup by specifying setOwnerNotCleanup = false. This results in the contract's native balance being sent directly to the initialOwner via selfdestruct. This behavior is problematic because the owner may not expect to receive native funds directly, or may be a contract that cannot process or forward native currency, leading to funds being lost or stuck. This opens up the protocol to griefing attacks, potentially even causing loss of user funds.
Recommendations	It is recommended to update the cleanup function such that any native funds held by the contract are first wrapped into WNATIVE before the contract is destroyed. This ensures that all value remains at the proxy address, accessible as an ERC-20 token.
Comments / Resolution	Fixed, the cleanup function now wraps any remaining native tokens instead of sending them to the owner.

Issue_03	Merkle tree can be used to obscure malicious transactions
Severity	**Low**
Description	The contract is primarily intended for single-use scenarios: deployment, token approval, swapping, and self-destruction. However, it is technically possible to repurpose it as a SmartWallet. This becomes problematic when a malicious actor constructs the Merkle tree, as users have no straightforward way to view the list of pre-signed transactions embedded in the tree. A malicious deployer could insert hidden approvals. If the user later reuses the contract as a SmartWallet, these hidden approvals could be exploited to steal funds.
Recommendations	Consider avoiding using these contracts as SmartWallets, especially if the Merkle root was not generated by the user. Consider providing toolings to let users inspect the Merkle tree: extract and decode all pre-signed transactions, reconstruct the Merkle root, and verify that it matches the one in the contract. This would help users confirm that no hidden or malicious operations are embedded.
Comments / Resolution	Fixed, the API response will contain all the information that the user would need to reconstruct or verify the Merkle Tree.

Issue_04	The _WNATIVE_STORAGE contract permanently locks native tokens upon self-destruction
Severity	**Informational**
Description	The _WNATIVE_STORAGE contract is used to allow the factory to retain the same address across different networks while referencing different WNative token addresses. After initialization, the deployer can call the contract again to self-destruct it and reclaim gas. However, it uses selfdestruct(address(this)), which sends any native token balance to the contract's own address—effectively locking the funds permanently. This issue has only been rated informational as this is not expected to cause issues since the contract is not meant to hold any native tokens.
Recommendations	Consider acknowledging this behavior as a known limitation. If native tokens were ever expected to be present, a safer alternative would be to transfer the balance to tx.origin or another recoverable address before calling selfdestruct, though this would require changing the contract's bytecode and reminting its vanity address.
Comments / Resolution	Acknowledged, the contract will not hold funds.

Issue_05	Risky use of returndatasize after a call
Severity	**Informational**
Description	The approvePermit2 function enables any user to authorize the Permit2 contract to spend their tokens, facilitating signature-based transfers without repeated approvals. For gas optimization, the contract uses returndatasize() instead of PUSH1 0x00 to set the selector of the approval function. This is safe as long as no external calls are made prior to this operation. However, when dealing with the native token, the function wraps it into the WNative token to interact as an ERC20. If the WNative contract returns any data during this wrapping process, it could alter the value of returndatasize(), leading to an incorrect slot and potentially rendering the approval logic unusable for native tokens. Although the constructor enforces a check to ensure that the WNative contract returns zero data during wrapping, this safeguard assumes the behaviour doesn’t change. If the WNative contract is upgradeable and is ever upgraded to return data, it could break compatibility. This issue is marked as informational due to its low likelihood and the presence of the constructor check.
Recommendations	To improve robustness and future-proof the contract, consider replacing returndatasize() with PUSH1 0x00 when setting the approval selector, especially if there's any chance of using a WNative implementation that may return data.
Comments / Resolution	Acknowledged, none of the WNative contracts intended for use return any data during wrapping operations. An upgrade that changes this behavior would most likely never happen.

Issue_06	Permissionless approvePermit2 function can enable denial-of-service on non-atomic swaps
Severity	**Informational**
Description	The approvePermit2 function is publicly accessible and allows anyone to approve any amount of tokens on behalf of a user. This introduces a potential denial-of-service (DoS) vector: An attacker could monitor the mempool for approval transactions and front-run them by calling approvePermit2 with a zero amount, effectively resetting the approval and causing the subsequent swap to fail. Additionally the approvePermit2 function can be used to forcefully wrap native tokens into WNative in order to DOS transferring of native tokens via a call to cleanup. However, this risk is mitigated if the contract is used as intended—with all actions (approval and swap) batched atomically via a Multicall. In such a case, the approval cannot be front-run, and the DoS vector is eliminated.
Recommendations	Consider always batching the approval and the action using it in a single atomic call, such as via the Multicall contract. This ensures the approval cannot be intercepted or modified before execution.
Comments / Resolution	Acknowledged, it is intended to use the Multicall contract while interacting with this contract.