C8NTINUUM Bridge Audit Report

Version 1.0

IhtishamSudo

December 29, 2025

C8NTINUUM Bridge

IhtishamSudo

December 29, 2025

Prepared by: IhtishamSudo

Auditor: - IhtishamSudo

Table of Contents

Protocol Summary

C8NTINUUM Bridge is a cross-chain bridging protocol that enables users to transfer assets between different blockchain networks, including Ethereum-compatible chains (Aeon) and Solana. The protocol utilizes a settlement-based architecture where:

- Users open orders on the source chain by depositing tokens (CTM)
- Fillers execute orders on the destination chain by providing the requested tokens
- A verifier validates cross-chain fills and settles orders
- The Settlement contract integrates with Uniswap V2 Router for token swaps

The bridge supports both standard token swaps and pledge tokens through a launchpad integration.

Disclaimer

This security audit was conducted by IhtishamSudo. The findings in this report are based on the code reviewed at the time of the audit and do not guarantee the absence of all vulnerabilities. This audit should not be considered as investment advice or an endorsement of the protocol. Users should conduct their own due diligence before interacting with the protocol.

Risk Classification

Likelihood	Impact	Severity
High	Critical	C
High	High	H
High	Medium	H/M
High	Low	M
Medium	Critical	H
Medium	High	H/M
Medium	Medium	M
Medium	Low	M/L
Low	Critical	M
Low	High	M

Likelihood	Impact	Severity
Low	Medium	M/L
Low	Low	L

Audit Details

Scope

The audit focused on the following components of the C8NTINUUM Bridge protocol:

- **Settlement.sol** - Core settlement contract handling order management, fills, and withdrawals
- **Cross-chain order hash generation** - Both Solidity and Rust implementations

Roles

- **User** - Opens cross-chain orders by depositing source tokens
- **Filler** - Executes orders on destination chains by providing requested tokens
- **Verifier** - Validates cross-chain fills and settles orders on the source chain
- **Owner/Admin** - Protocol administrator with privileged access

Executive Summary

The security audit of the C8NTINUUM Bridge protocol revealed **5 findings** across various severity levels. The most significant issue identified is a race condition that allows users to double-spend by receiving tokens on the destination chain while withdrawing their original deposit on the source chain.

Key areas of concern include: - Cross-chain atomicity and settlement timing - Order hash integrity across chains - ETH handling and fund recovery mechanisms - View function correctness

Issues found

Severity	Count
Critical	0

Severity	Count
High	1
Medium	3
Low	1
Informational	0
Total	**5**

Findings

Critical

No critical findings.

High

[H-1] Race Condition Allows User to Receive Tokens on Destination Chain AND Withdraw Original Deposit

Description

The cross-chain fill flow has no atomicity guarantee. The `settle()` and `withdraw()` functions have mutual exclusion checks, but there is a timing gap that creates a race condition:

```
// settle()
require(orders[orderId].withdrawn == false, "Already withdrawn");

// withdraw()
require(orders[orderId].settled == false, "Already settled");
```

The `withdraw()` function only checks: - Order exists - Order not settled - Order not withdrawn - Deadline has passed

It does **not** check if the order was already filled on the destination chain.

Impact

A user can receive tokens on the destination chain (Solana) AND withdraw their original deposit on the source chain (Aeon), effectively double-spending and stealing funds from the filler.

Proof of Concepts

```
T=0       User opens order on Aeon: depositAmount=1000 CTM, fillDeadline
    =T+10min

T+5min    Filler fills on Solana -> User receives destination tokens
          FillEvent emitted, user has destination tokens

T+15min   Deadline + withdrawDeadlineDuration passes
          Verifier hasn't settled yet

T+15min   User calls withdraw() on Aeon
          settled=false, withdrawn=false, time passed
          User receives 1000 CTM back!

T+16min   Verifier tries to settle()
          FAILS: "Already withdrawn"
```

Result: User now has BOTH: - Destination tokens on Solana - Original 1000 CTM withdrawn on Aeon

Recommended Mitigation

Add a check in the `withdraw()` function to verify the order was not already filled:

```
function withdraw(bytes32 orderId) external {
    require(isOrderFilled[orderId] == false, "Order already filled");
    // ... existing checks
}
```

Team Response

Acknowledged. Will Consider it

Medium

[M-1] Order Hash Excludes `destination_chain_id`, Allowing Orders to be Filled on Unintended Chains

Description

The order hash calculation in both the Solidity and Rust implementations excludes the `dstChainId` parameter. This allows the same order hash to be valid across multiple destination chains.

Solidity:

```
orderId = keccak256(abi.encode(
    block.chainid, address(this), msgSender, amount,
    nonce, dstAddress, dstToken, dstAmount
    // Missing: dstChainId
));
```

Rust:

```
let digest = hashv(&[
    src_chain_id, src_ca, src_user, amount,
    order_index, dest_recipient, dest_token, dest_amount
    // Missing: dst_chain_id
]);
```

Impact

A malicious filler can execute an order on an unintended chain. Funds may be delivered to the wrong chain or permanently lost if the destination address is a chain-specific contract.

Proof of Concepts

1. User opens order: Ethereum -> Aeon (intended destination)
2. Malicious filler executes the order on Optimism instead
3. Same hash validates successfully (no chain ID check)
4. Funds delivered to wrong chain or permanently lost

Recommended Mitigation

Include `dstChainId` in both hash calculations:

```
orderId = keccak256(abi.encode(..., openOrder.dstChainId));
```

Team Response

Fixed

[M-2] Settlement Contract Fails to Handle ETH Refunds Returned by Uniswap Router

Description

When `fillAeon` executes a swap for a non-pledge token (or a graduated one), it calls `router.swapExactETHForTokens`. The Uniswap V2 Router refunds any unused ETH (dust) if the exact output amount requires less ETH than provided.

Settlement.sol - fillAeon:

```
if (amountOut >= fillAmount) {
    if(isPledge) {
        launchpad.swapExactETHForTokens{value: inputAmount}(fillAmount,
            path, resolvedDstAddress, block.timestamp);
    } else {
        router.swapExactETHForTokens{value: inputAmount}(fillAmount,
            path, resolvedDstAddress, block.timestamp); // <--
    }
}
```

The Router calculates the exact ETH needed (`amounts[0]`) to buy the requested `fillAmount`. If `amounts[0] < inputAmount`, the Router refunds the difference to `msg.sender` (the Settlement contract).

The Settlement contract has a `receive()external payable {}` function, so it accepts the refunded ETH. However, there is no admin sweep or withdraw function for excess ETH, only user-specific withdrawal logic for unfilled orders.

Impact

Any ETH returned by the Router accumulates in the Settlement contract forever and is permanently locked. This balance will keep increasing to a significant amount over time.

Proof of Concepts

1. Alice wants to bridge tokens to buy USDT (non-pledge) on Aeon. She initiates an order that translates to 10 CTM on Aeon
2. Bob (Filler) calls `fillAeon` sending 10 CTM
3. The contract calculates `inputAmount = 10 CTM`
4. It calls `router.swapExactETHForTokens{value: 10 CTM}` requesting 100 USDT
5. The current market rate is 1 CTM = 11 USDT
6. To get 100 USDT, the Router only needs ~9.09 CTM
7. The Router executes the swap:
   - Takes 9.09 CTM
   - Sends 100 USDT to Alice
   - Refunds 0.91 CTM to the Settlement contract
8. The Settlement contract receives the 0.91 CTM
9. Since there is no function to access this balance, the 0.91 CTM is locked forever

Recommended Mitigation

Add an admin withdraw function to sweep accumulated ETH funds:

```
function sweepETH(address payable recipient) external onlyOwner {
    uint256 balance = address(this).balance;
    require(balance > 0, "No ETH to sweep");
    recipient.transfer(balance);
}
```

Team Response

Will Fix

[M-3] Unintended ETH Acceptance in `fill` Function Leads to Permanent Loss of Funds

Description

The `fill` function in `Settlement.sol` is declared as `payable`, allowing fillers to send native ETH with their transactions. However, the function logic never reads, validates, or uses `msg.value` anywhere in its execution flow. The function operates exclusively by pulling CTM tokens from the filler via `safeTransferFrom` and performing token swaps through the Uniswap router.

```
function fill(...) external payable {
    // msg.value is never used or validated
    // Only CTM tokens are pulled via safeTransferFrom
    CTM.safeTransferFrom(msg.sender, address(this), inputAmount);
    // ...
}
```

Impact

Any ETH accidentally sent by fillers will remain stuck in the Settlement contract's balance with no recovery mechanism, leading to permanent loss of funds.

Proof of Concepts

1. Filler calls `fill()` with `msg.value = 1 ETH` (accidentally or due to UI bug)
2. The function accepts the ETH due to `payable` modifier
3. Function only uses `safeTransferFrom` to pull CTM tokens
4. The 1 ETH sent is never used and remains in the contract
5. No withdrawal mechanism exists for this ETH, funds are permanently lost

Recommended Mitigation

Either remove the `payable` modifier or add an explicit check to reject ETH:

Option 1 - Remove payable:

```
function fill(...) external { // Remove payable
    // ...
}
```

Option 2 - Explicit check:

```
function fill(...) external payable {
    require(msg.value == 0, "ETH not accepted");
    // ...
}
```

Team Response

Fixed

Low

[L-1] Incorrect Indexing in `getUserOrdersInterval` Function Returns Wrong Order Data

Description

The `getUserOrdersInterval` function uses incorrect indexing where it always starts from index `i=0` instead of the specified `left` parameter. This causes the function to return incorrect order data from the wrong index range.

Impact

When users request a specific range of orders (e.g., orders 5-10), the function returns orders from the wrong range (orders 0-5 instead).

Proof of Concepts

When calling `getUserOrdersInterval(user, 5, 10)`: - **Expected behavior**: Returns orders at indices 5, 6, 7, 8, 9, 10 - **Actual behavior**: Returns orders at indices 0, 1, 2, 3, 4, 5

Recommended Mitigation

Adjust the indexing in the for loop to use `left + i` instead of just `i`:

```
function getUserOrdersInterval(address user, uint256 left, uint256
    right) external view returns (CrossChainOrder[] memory) {
     //....SNIP

     for (uint256 i = 0; i < length; i++) {
         ans[i] = orders[userOrders[user][left + i]]; // Fixed indexing
     }
     return ans;
}
```

Team Response

Fixed

Informational

No informational findings.