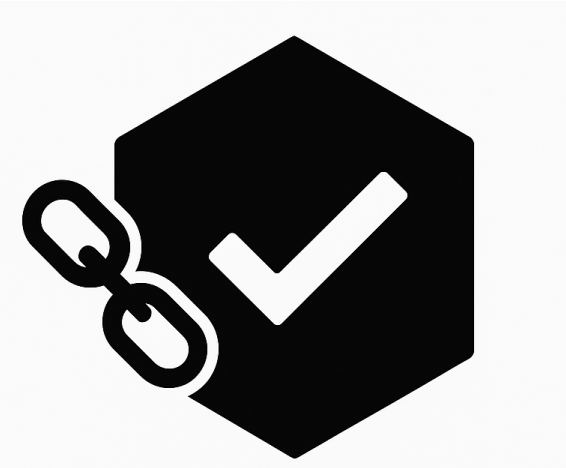

Boss Bridge Protocol Audit Report

Version 1.0

github.com/lucasfhope

October 13, 2025

Boss Bridge Protocol Audit Report

Lucas Hope

October 13, 2025

Prepared by: Lucas Hope

Table of Contents

Protocol Summary

The Boss Bridge protocol intends to be a simple bridge mechanism to move an ERC20 from L1 to an L2. The bridge allows users to deposit tokens into a vault on the L1, which will trigger an event that off-chain operators pick up, who will then will mint the corresponding tokens on the L2. The L2 contracts are still being built, but there will be a way to bridge tokens back to the L1 where they can be withdrawn from the vault.

Disclaimer

Lucas Hope 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

The findings described in this report corresponds to the following commit hash:

```
d59479cbf925f281ef79ccb35351c4ddb002c3ef
```

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
- 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

The audit found some major vulnerabilities, issues with the design of the bridge protocol, and the centralization risk coming from the power of the signers/operators. The audit also found that the off-chain logic must be implemented correctly to ensure the safety of the protocol.

Issues found

Severity	Number of issues found
High	6
Medium	2
Low	1
Info	3
Gas	4
Total	**16**

Findings

High

[H-1] Arbitrary from in transferFrom in `L1BossBridge::depositTokensToL2` allows an attacker to deposit tokens from another user that has already approved tokens to be bridged.

Description: `L1BossBridge::depositTokensToL2` takes in a `from` address to be used as the user to send tokens to the bridge. If an unsuspecting user approves their tokens to bridge, then it is possible for an attacker to call `depositTokensToL2` before the user with the user's address as the `from` address.

Impact: An attacker can steal another users tokens and send the tokens to their own address on the layer 2.

Proof of Concept:

1. A user approves tokens for the Boss Bridge contract.
2. An attacker notices the approval and calls the `depositTokensToL2` function with the user as the `from` address and the attacker as the `to` address for the approved amount.

Proof of Code: Add the following test to the test suite in `L1TokenBridge.t.sol`.

```
function testCanMoveApprovedTokensOfOtherUsers() public {
    // eventual attacked user approves the bridge to spend their tokens
    vm.prank(user);
    token.approve(address(tokenBridge), type(uint256).max);

    // attacker can now move the user's tokens to the vault and send
       them to themselves on L2
    uint256 depositAmount = token.balanceOf(user);
    address attacker = makeAddr("attacker");
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(user, attacker, depositAmount);
    tokenBridge.depositTokensToL2(user, attacker, depositAmount);

    assert(token.balanceOf(user) == 0);
    assert(token.balanceOf(address(vault)) == depositAmount);
}
```

The test will show that the protocol will take the tokens from the user and emit the deposit such that the attacker will receive the bridged tokens on the L2.

Recommended Mitigation: Remove the ability to set the `from` address in `depositTokensToL2` and hard code `msg.sender` as the from address in `safeTransferFrom`. Remeber to update the

emit as well.

```
-    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)

         // 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] Arbitrary from in transferFrom in `L1BossBridge::depositTokensToL2` and approval of all vault tokens for the `L1TokenBridge` contract in the constructor allows an attacker to mint unbacked tokens on the L2.

Description: `L1BossBridge::depositTokensToL2` takes in a `from` address to be used as the user to send tokens to the bridge. Since the vault has already approved `type(uint256).max` tokens for the `L1TokenBridge` contract in the constructor, an attacker could pass in the vault address as the `from` address and continuously emit a deposit to their address on the L2. This could allow for the minting of tokens on the L2 that are not backed by tokens locked in the L1 vault.

Impact: Tokens could be minted on the L2 that are not backed by tokens locked in the L1 vault.

Proof of Concept: Any user can send a transaction where the `depositTokensToL2` function is called with the vault address as the `from` parameter.

Proof of Code Add the following test to the test suite in `L1TokenBridge.t.sol`.

```
function testCanTransferFromVaultToVault() public {
    uint256 depositAmount = 100e18;
    deal(address(token), address(vault), depositAmount);

    address attacker = makeAddr("attacker");
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), attacker, depositAmount);
    tokenBridge.depositTokensToL2(address(vault), attacker, depositAmount);
    vm.stopPrank();
```

```
11 }
```

The test will show that the transaction will go through and the deposit event will be emitted with the recipient as the attacker.

Recommended Mitigation: The `from` address should be removed as a parameter in `depositTokenToL2` and `msg.sender` should be hard coded as the `from` address, which is shown in the previous finding. This is the best option. Other mitagations that can be considered but are not neccessary:

- Remove the blanket approval of the bridge to transfer tokens from the vault in `L1TokenBrige::sendToL1`. Instead, this can be done only when withdrawing in `sendToL1`.
- The off chain process that reads the deposit events could be programmed to disregard deposits where the `from` address is the vault address.

[H-3] Revealing the `v`, `r`, and `s` values of a signature on chain in `L1TokenBridge::withdrawTokensToL1` and `L1TokenBridge::sendToL1` make it possible for a replay attack.

Description: By exposing the signature on chain in `withdrawTokensToL1` and `sendToL1`, attackers can reuse the signature for the same recipient and token amount. This is possible because the function lacks any fields that would prevent the same from signature being used repeatedly.

Impact: An attacker could drain the funds in the vault by reusing the signature to take out multiple withdrawals.

Proof of Concept:

1. An attacker deposits tokens in the `L1TokenBridge` contract.
2. The attacker uses the bridge unlock the tokens in the L1 and waits for an operator to sign the transaction an call `withdrawTokensToL1`.
3. The attacker uses the signature to continously call `withdrawTokensToL1` and drain the vault of its tokens.

Proof of Code: Add the following test to the test suite in `L1TokenBridge.t.sol`.

```
function testSignatureReplay() public {
    address attacker = makeAddr("attacker");
    // vault already holds some tokens
    uint256 vaultInitialBalance = 10000e18;
    uint256 attackerInitialBalance = 100e18;
    deal(address(token), address(vault), vaultInitialBalance);
    deal(address(token), attacker, attackerInitialBalance);

    // attacker deposits tokens
```

```
10      vm.startPrank(attacker);
11      token.approve(address(tokenBridge), type(uint256).max);
12      tokenBridge.depositTokensToL2(attacker, attacker,
            attackerInitialBalance);
13      vm.stopPrank();
14
15      // signer/operator signs the withdrawal
16      bytes memory message = abi.encode(address(token), 0, abi.encodeCall
            (IERC20.transferFrom, (address(vault), attacker,
            attackerInitialBalance)));
17      (uint8 v, bytes32 r, bytes32 s) = vm.sign(
18          operator.key,
19          MessageHashUtils.toEthSignedMessageHash(keccak256(message))
20      );
21      tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance, v,
            r, s);
22
23      // attacker sees the signed message on chain and replays to steal
            tokens from the vault
24      // vm.prank(attacker);   // This would be coming from the attacker
25      while(token.balanceOf(address(vault)) > 0) {
26          tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance
                , v, r, s);
27      }
28
29      assert(token.balanceOf(address(vault)) == 0);
30      assert(token.balanceOf(attacker) == attackerInitialBalance +
            vaultInitialBalance);
31  }
```

The test shows that an attacker can reuse a signature to perform multiple withdrawals that can drain the vault.

Recommended Mitigation: Consider implementing a feature that includes an incrementing nonce value in the encoded message which can not be reused to withdraw tokens from the protocol.

[H-4] CREATE opcode does not work on zkSync Era.

Description: `TokenFactory::deployToken` uses the CREATE opcode to deploy ERC20 token contracts. However, this does not work on zkSync Era. This is well documented here.

Impact: If deployed on zkSync, `TokenFactory` will not be able to deploy any contracts.

Recommended Mitigation: Update `TokenFactory` so that it can deploy contracts on zkSync Era. This will require using the zkSync deployer system contract.

[H-5] `L1BossBridge::withdrawalTokensToL1` does not validate that the withdrawal amount matches the amount deposited, allowing an attacker to withdraw more than they deposited.

Description: The protocol currently does not track the deposits and withdrawals of individual users. This means that the handling of the withdrawal amounts are up to the bridge operators/signers.

Impact: Incorrect off-chain operations can risk funds being mishandled.

Recommended Mitigation: Ensure that the L2 mints/unlocks tokens based on the deposit amount emitted from the L1, and, likewise, the L1 unlocks tokens based solely on deposit events emitted from the L2. Assuming the other vulnerabilities are fized, operators should rely on the contract events and not sign other messages.

[H-6] `TokenFactory` is minted tokens but has no way of transferring them out, so no users will receive tokens and no one will be able to bridge them.

Description: `TokenFactory::deployToken` deploys an `L1Token`, which mints an initial amount to `msg.sender`, which is the `TokenFactory` contract. However, the `TokenFactory` contract does not have the ability to call approve on its tokens or transfer them to any other address.

Impact: No one will be able to receive any tokens, which will make the protocol useless.

Recommended Mitigation: It is possible to add a transfer function with an `onlyOwner`.

This issue, however, calls into question the design of the protocol. The `TokenFactory` contract is nearly useless other than tracking tokens that have been deployed. It is likely that the bridge contract is more important to track, which will be important to know how to bridge the tokens, but it will also have the token address to read from the bytecode. You could have a `BridgeFactory` contract that deploys the token and deploys the bridge in one transaction, and it can map the bridge to the token.

Regardless, this process should be handled in one transaction rather than multiple. If you do not us a `BridgeFactory`, you should at least use a deploy script to perform the deploys in one transaction.

Medium

[M-1]: Centralization and unvetted signer privileges potentially put protocol funds at risk.

Description: Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds. Furthermore, if the owner loses their private key or it falls into the hands of an attacker, control of the protocol and its funds will be lost.

More importantly, operators/signers that the owner can set have limitless access to send tokens to any address in `L1TokenBridge::withdrawTokensToL1` and can perform unprotected calls in `L1TokenBridge::sendToL1`.

Recommended Mitigation: Consider marking `sendToL1` as an `internal` function to limit the external calls operators/signers can make. This will also prevent an operator from signing a malicious request and sending the transaction directly through the unprotected `sendToL1` function.

Nevertheless, `withdrawTokensToL1` can be used by operators to send tokens from the vault to any address. Therefore, make sure only trusted bridge operators are given signer privileges. Operators will have to and ensure that that the withdraw requests are of the correct amount and to the correct address, as is specified when a user submits a deposit to bridge tokens from the L2 to the L1.

[M-2] `L1BossBridge::depositTokensToL2` vault token balance check against `DEPOSIT_LIMIT` allows for a denial of service attack.

Description: By using the `token.balanceOf(address(vault))` to check the `DEPOSIT_LIMIT`, the `L1BossBridge` contract is succeptable to a denial of service attack. If tokens are sent to the vault without going through `L1BossBridge::depositTokensToL2`, the tokens will counted towards the limit even though they will not be bridged to the L2. This does require an attacker to receive protocol tokens and send them to the vault, effectively reliquishing the value of the tokens. However, if this happens, the protocol will have a smaller deposit limit than intended.

Impact: The protocol will have a smaller deposit limit, and it could potentially prevent users from using the bridge.

Proof of Concept:

1. An attacker has a number of tokens, which could be over the 100,000 token deposit limit.
2. The attacker sends the tokens directly to the vault contract. This will reduce the amount of tokens that can be deposited to be bridged to the L2. If the amount of tokens sent to the vault is greater than 100,000 tokens, the `L1BossBridge::depositTokensToL2` function will always revert.

Proof of Code: Add the following test to the test suite in `L1TokenBridge.t.sol`.

```
function testDenialOfService() public {
        address attacker = makeAddr("attacker");
        deal(address(token), attacker, 100000e18);

        vm.startPrank(attacker);
        token.transfer(address(vault), 100000e18);
        vm.stopPrank();

```

```
9           vm.startPrank(user);
10          token.approve(address(tokenBridge), type(uint256).max);
11          vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
               selector);
12          tokenBridge.depositTokensToL2(user, user, 1e18);
13          vm.stopPrank();
14      }
```

The test shows that an attacker can send 100,000 tokens to the vault and deny further use of `depositTokensToL2`.

Recommended Mitigation: `L1BossBridge` should do its own accounting of tokens during deposits and withdrawals rather than relying on the token balance of the vault.

Low

[L-1] `TokenFactory::deployToken` does not check the symbol, allowing for tokens with duplicate symbols to overwrite entries in `TokenFactory:s_tokenToAdress`.

Description: The `deployToken` function takes a `symbol` as a parameter but does not check if there is an entry in `s_tokenToAddress`. Therefore, the new token can overwrite another token in 's_tokenToAddress.

Recommended Mitigation:

```
1       function deployToken(string memory symbol, bytes memory
           contractBytecode) public onlyOwner returns (address addr) {
2   +       if(s_tokenToAddress != address(0))
3   +           revert();
4   +       }
5           assembly {
6               addr := create(0, add(contractBytecode, 0x20), mload(
                   contractBytecode))
7           }
8           s_tokenToAddress[symbol] = addr;
9           emit TokenDeployed(symbol, addr);
10      }
```

Informational

[I-1] No deploy scripts

Consider writing deploy scripts to make deploying contracts easier and ensures the deployments are in the proper order.

[I-2] `L1BossBridge:depositTokentoL2` should emit the deposit event before the external token transfer to follow best practices.

Functions in the protocol should follow CEI to adhere to the best practices. Therefore, `depositTokenToL2` should call `safeTransferFrom` after emitting the event.

```
    function depositTokensToL2(address from, address l2Recipient,
       uint256 amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }

-       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);

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

[I-3] State Change Without Event

There are state variable changes in this function but no event is emitted. Consider emitting an event to enable offchain indexers to track the changes.

3 Found Instances

- Found in src/L1BossBridge.sol Line: 57

```
    function setSigner(address account, bool enabled) external
       onlyOwner {
```

- Found in src/L1BossBridge.sol Line: 92

```
    function withdrawTokensToL1(address to, uint256 amount, uint8
       v, bytes32 r, bytes32 s) external {
```

- Found in src/L1BossBridge.sol Line: 113

```
1      function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
          message) public nonReentrant whenNotPaused {
```

Gas

[G-1] Withdrawals are prone to undboanded gas consumption due to return bombs

Description: In `L1BossBridge::sendToL1`, the bridge executes a low-level call to an arbitrary target passing all available gas to the caller, which is presumably the signer. 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, specifically signers, 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.

Recommended Mitigation: 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.

Also, consider marking `sendToL1` as `internal`. This will force users and operators to only go through `L1BossBridge::withdrawTokensToL1`, which contains a predefined messages that will call `IERC20.transferFrom`. Therefore, users will only be able to receive a working signature for a valid withdraw request, and operators will only be able to sign an execute valid withdraw requests. Operators will still have to ensure that the withdraw requests are of the correct amount and to the correct address, as is specified when a user submits a deposit to bridge tokens from the L2 to the L1.

[G-2] Public Function Not Used Internally

If a function is marked public but is not used internally, consider marking it as `external` to save gas on deployment and when called.

2 Found Instances

- Found in src/TokenFactory.sol Line: 23

```
1      function deployToken(string memory symbol, bytes memory
          contractBytecode) public onlyOwner returns (address addr) {
```

- Found in src/TokenFactory.sol Line: 31

```
1       function getTokenAddressFromSymbol(string memory symbol)
            public view returns (address addr) {
```

[G-3]: State Variable Could Be Immutable

State variables that are only changed in the constructor should be declared immutable to save gas. Add the `immutable` attribute to state variables that are only changed in the constructor

1 Found Instances

- Found in src/L1Vault.sol Line: 13

```
1       IERC20 public token;
```

[G-4] State Variable Could Be Constant

State variables that are not updated following deployment should be declared constant to save gas. Add the `constant` attribute to state variables that never change.

1 Found Instances

- Found in src/L1BossBridge.sol Line: 30

```
1       uint256 public DEPOSIT_LIMIT = 100_000 ether;
```