Boss Bridge Audit Report

Version 1.0

Lokapal

February 8, 2025

Boss Bridge Audit Report

Lokapal

February 5, 2025

Prepared by:

- Lokapal

Lead Auditor:

- Ricardo Pintos

Table of Contents

    * [H-3] `L1BossBridge::constructor` approves the max amount of token transfers, which allows users to mint unlimited tokens to the L2
    * [H-4] `L1BossBridge::withdrawTokensToL1` does not check for signature replay
  - Low
    * [L-1] `L1Vault::approveTo` ignores return value
    * [L-2] Missing checks in `L1Vault::constructor` for `address(0)` when assigning values to address state variables
    * [L-3] Unsafe ERC20 Operations should not be used
    * [L-4]: In `L1BossBridge::setSigner`, state variable changes but no event is emitted
    * [L-5] Low level call in `L1BossBridge::sendToL1`
  - Informational
    * [I-1] Solidity `pragma` should be specific, not wide
    * [I-2]: State variable could be declared constant in `L1BossBridge`
    * [I-3]: State variable could be declared immutable in `L1Vault`
    * [I-4] In `TokenFactory`, `public` functions not used internally could be marked `external`
    * [I-5] Events with missing `indexed` fields
    * [I-6] PUSH0 is not supported by all chains

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

The LOKAPAL 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
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

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.

Executive Summary

This security review was conducted as part of Cyfrin Updraft's Smart Contract Security course.

Issues found

Severity	Number of issues found
High	4
Medium	0
Low	5
Info	6
Total	15

Findings

High

[H-1] `TokenFactory::deployToken` uses `create` and will not work with zkSync

Description: The `deployToken` function uses the low level `create` command. This approach will save gas, but raises multichain deployment issues:

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

```
2       assembly {
3   ~~>     addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
4       }
5       s_tokenToAddress[symbol] = addr;
6       emit TokenDeployed(symbol, addr);
7   }
```

Impact: In addition to Ethereum `Mainnet`, the protocol will deploy to `zkSync`. This chain doesn't support `create` for deployment. You can read the docs here.

Proof of Concept: 1. The protocol starts building its infrastructure on `Mainnet`, 2. Eventually, the `TokenFactory.sol` contract has to be deployed on `zkSync`, 3. Deployment fails, leaving the protocol without its intended L2 platform.

Recommended mitigation: Unless there are additional considerations other than gas cost, consider following a deployment process compatible with both `Mainnet` and `zkSync`.

[H-2] `L1BossBridge::depositTokensToL2` uses an arbitrary from in `transferFrom`

Description: The `depositTokensToL2` function does not check the if the msg.sender is the address that will approve the transfer of tokens.

```
1   function depositTokensToL2(address from, address l2Recipient, uint256 amount)
        external whenNotPaused {
2       if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
3           revert L1BossBridge__DepositLimitReached();
4       }
5   ~~> token.safeTransferFrom(from, address(vault), amount);
6
7       // Our off-chain service picks up this event and mints the corresponding
            tokens on L2
8       emit Deposit(from, l2Recipient, amount);
9   }
```

Impact: This means that after a user approves the transfer of their tokens, anyone can use that approval to transfer to themselves all the amount approved. They just have to monitor approvals and front-run the legitimate owner of the tokens.

Proof of Concept: Run this test on your `L1TokenBridge.t.sol` unit suite:

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

```
4
5       uint256 depositAmount = token.balanceOf(user);
6       address attacker = makeAddr("attacker");
7       vm.prank(attacker);
8       vm.expectEmit(address(tokenBridge));
9       emit Deposit(user, attacker, depositAmount);
10      tokenBridge.depositTokensToL2(user, attacker, depositAmount);
11
12      assertEq(token.balanceOf(user), 0);
13      assertEq(token.balanceOf(address(vault)), depositAmount);
14      vm.stopPrank();
15  }
```

Recommended mitigation: Consider adding a `msg.sender` check that matches the `from` parameter:

```
1  function depositTokensToL2(address from, address l2Recipient, uint256 amount)
       external whenNotPaused {
2  +    if (msg.sender != from) revert L1BossBridge__Unauthorized();
3      if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
4          revert L1BossBridge__DepositLimitReached();
5      }
6      token.safeTransferFrom(from, address(vault), amount);
7
8      // Our off-chain service picks up this event and mints the corresponding
           tokens on L2
9      emit Deposit(from, l2Recipient, amount);
10 }
```

[H-3] `L1BossBridge::constructor` approves the max amount of token transfers, which allows users to mint unlimited tokens to the L2

Description: The `constructor` calls the `approveTo` function setting an extremely high amount of preapproved tokens.

```
1  constructor(IERC20 _token) Ownable(msg.sender) {
2      token = _token;
3      vault = new L1Vault(token);
4      // Allows the bridge to move tokens out of the vault to facilitate
           withdrawals
5  ~~> vault.approveTo(address(this), type(uint256).max);
6  }
```

Impact: This means that after the `constructor` approves the transfer of tokens, anyone can use that approval to transfer to themselves all the amount approved. They can mint an unlimited amount of tokens, disrupting the balance between the L1 and L2 chains.

Proof of Concept: Run this test on your `L1TokenBridge.t.sol` unit suite:

```
function testCanTransferFromVaultToVault() 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);
}
```

Recommended mitigation: Consider adding a `msg.sender` check that matches the `from` parameter:

```
function depositTokensToL2(address from, address l2Recipient, uint256 amount)
    external whenNotPaused {
+   if (msg.sender != from) revert L1BossBridge__Unauthorized();
    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);
}
```

[H-4] `L1BossBridge::withdrawTokensToL1` does not check for signature replay

Description: The `v, r and s` values are stored on-chain when calling `L1bossBridge::sendToL1`:

```
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();
```

```
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      }
```

And when the signer withdraw tokens back to the L1, there are no checks to prevent using again the same signature:

```
1   function withdrawTokensToL1(address to, uint256 amount, uint8 v, bytes32 r,
        bytes32 s) external {
2           sendToL1(
3               v,
4               r,
5               s,
6               abi.encode(
7                   address(token),
8                   0, // value
9   ~~>             abi.encodeCall(IERC20.transferFrom, (address(vault), to, amount
        ))
10              )
11          );
12      }
```

Impact: Anyone can execute a signature replay attack after the signature is on-chain, disrupting the token balance between the L1 and L2 chains.

Proof of Concept: Run this test on your `L1TokenBridge.t.sol` unit suite:

```
1   function testSignatureReplay() public {
2       address attacker = makeAddr("attacker");
3       uint256 vaultInitialBalance = 1000e18;
4       uint256 attackerInitialBalance = 100e18;
5       deal(address(token), address(vault), vaultInitialBalance);
6       deal(address(token), address(attacker), attackerInitialBalance);
7
8       vm.startPrank(attacker);
9       token.approve(address(tokenBridge), type(uint256).max);
10      tokenBridge.depositTokensToL2(attacker, attacker, attackerInitialBalance);
11
12      bytes memory message = abi.encode(address(token), 0, abi.encodeCall(IERC20.
            transferFrom, (address(vault), attacker, attackerInitialBalance)));
```

```
13      (uint8 v, bytes32 r, bytes32 s) = vm.sign(operator.key, MessageHashUtils.
            toEthSignedMessageHash(keccak256(message)));
14
15      while(token.balanceOf(address(vault)) > 0){
16          tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance, v, r,
                s);
17      }
18
19      assertEq(token.balanceOf(attacker), attackerInitialBalance +
            vaultInitialBalance);
20      assertEq(token.balanceOf(address(vault)), 0);
21      vm.stopPrank();
22  }
```

Recommended mitigation: Consider adding a nonce (number only used once) or a deadline parameter in the `sendToL1` function, so the signature can't be used multiple times.

Low

[L-1] `L1Vault::approveTo` ignores return value

Description: The `approveTo` function ignores return value by `token.approve`:

```
1  function approveTo(address target, uint256 amount) external onlyOwner {
2      token.approve(target, amount);
3  }
```

Recommended mitigation: Ensure that all the return values of the function calls are used.

[L-2] Missing checks in `L1Vault::constructor` for `address(0)` when assigning values to address state variables

Description: The `_token` parameter is not checked for `address(0)` in the constructor:

```
1      constructor(IERC20 _token) Ownable(msg.sender) {
2          token = _token;
3      }
```

Recommended mitigation: Check for `address(0)` when assigning values to address state variables.

```
// ERRORS
+   error L1Vault__NotAddressZero();

// FUNCTIONS
    constructor(IERC20 _token) Ownable(msg.sender) {
+       if (address(token) == address(0)) revert L1Vault__NotAddressZero();
        token = _token;
    }
```

[L-3] Unsafe ERC20 Operations should not be used

Description: ERC20 functions may not behave as expected. For example: return values are not always meaningful.

At `L1BossBridge`:

```
abi.encodeCall(IERC20.transferFrom, (address(vault), to, amount))
```

At `L1Vault`:

```
token.approve(target, amount);
```

Recommended mitigation: It is recommended to use `OpenZeppelin's SafeERC20` library.

[L-4]: In `L1BossBridge::setSigner`, state variable changes but no event is emitted

Description: State variable changes in this function but no event is emitted.

```
    function setSigner(address account, bool enabled) external onlyOwner {
        signers[account] = enabled;
    }
```

Recommended mitigation: Consider adding the corresponding event.

```
// EVENTS
+   event L1BossBridge__SignerEnabled(address account);

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

```
6 +         emit L1BossBridge__SignerEnabled(account);
7           signers[account] = enabled;
8       }
```

[L-5] Low level call in `L1BossBridge::sendToL1`

Description: The use of low-level calls is error-prone. Low-level calls do not check for code existence or call success.

```
1  (bool success,) = target.call{ value: value }(data);
```

Recommended mitigation: Avoid low-level calls. Check the call success. If the call is meant for a contract, check for code existence.

Informational

[I-1] Solidity `pragma` should be specific, not wide

Description: Multiple Solidity versions in different contracts can cause conflicts in functionality, specially with version `0.8.20`.

Version constraint `0.8.20` contains known severe issues - VerbatimInvalidDeduplication - FullInliner-NonExpressionSplitArgumentEvaluationOrder - MissingSideEffectsOnSelectorAccess.

Recommended mitigation: Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.20;`, use `pragma solidity 0.8.20`.

[I-2]: State variable could be declared constant in `L1BossBridge`

Description: State variables that are not updated following deployment should be declared constant to save gas.

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

Recommended mitigation: Add the `constant` attribute to state variables that never change.

```
+   uint256 public constant DEPOSIT_LIMIT = 100_000 ether;
-   uint256 public DEPOSIT_LIMIT = 100_000 ether;
```

[I-3]: State variable could be declared immutable in `L1Vault`

Description: State variables that are set in the constructor should be declared immutable to save gas:

```
    IERC20 public token;

    constructor(IERC20 _token) Ownable(msg.sender) {
        token = _token;
    }
```

Recommended mitigation: Add the `immutable` attribute to state variables that are only changed in the constructor:

```
+   IERC20 public immutable i_token;
-   IERC20 public token;

    constructor(IERC20 _token) Ownable(msg.sender) {
+       i_token = _token;
-       token = _token;
    }
```

[I-4] In `TokenFactory`, `public` functions not used internally could be marked `external`

Description: Functions that are not called internally should be declared `external` to save gas.

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

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

Recommended mitigation: Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

[I-5] Events with missing `indexed` fields

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields).

At `L1BossBridge`:

```
1 event Deposit(address from, address to, uint256 amount);
```

At `TokenFactory`:

```
1 event TokenDeployed(string symbol, address addr);
```

Recommended mitigation: Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, every field should be indexed.

[I-6] PUSH0 is not supported by all chains

Description: `Solc` compiler version `0.8.20` switches the default target EVM version to Shanghai, which means that the generated `bytecode` will include PUSH0 opcodes.

Recommended mitigation: Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than `mainnet` like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.