Boss Bridge Protocol Security Review

Reviewer: Muhammad Faran **Date:** July 14, 2025 **Repo:** Cyfrin/7-Boss-Bridge

---

Table of Contents

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and view of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk-Classification

The severity of each finding combines **Impact** (the damage if exploited) and **Likelihood** (the chance of exploitation). The matrix below illustrates how these two axes map to a single severity label:

Likelihood ↓ / Impact →	High	Medium	Low
High	High	High/Medium	Medium
Medium	High/Medium	Medium	Medium-Low
Low	Medium	Medium-Low	Low

Executive Summary

Severity	Number Of Issues Found
High	5
Medium	0
Low	0

Audit Details

The findings described in this document correspond to the following **commit hash**:

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

Scope

```
#-- src
|   #-- L1BossBridge.sol
|   #-- L1Token.sol
|   #-- L1Vault.sol
|   #-- TokenFactory.sol
```

Protocol Summary

The Boss Bridge is a bridging mechanism to move an ERC20 token (the "Boss Bridge Token" or **BBT**) from L1 to an L2 the development team claims to be building. Because the L2 part of the bridge is still under construction, it was not included in the reviewed codebase.

The bridge is intended to allow users to deposit tokens, which are to be held in a vault contract on L1. Successful deposits should trigger an event that an off-chain mechanism is in charge of detecting, in order to mint the corresponding tokens on the L2 side of the bridge.

Withdrawals must be approved by operators (or "signers"). Essentially, they are expected to be one or more off-chain services where users request withdrawals. These services should verify requests before signing the data users must use to withdraw their tokens.

It is worth highlighting that there's little-to-no on-chain mechanism to verify withdrawals, other than the operator's signature. Therefore, the Boss Bridge heavily relies on having robust, reliable, and always-available operators to approve withdrawals. Any rogue operator or compromised signing key may put at risk the entire protocol.

Roles

Bridge Owner

- Can pause and unpause withdrawals in the `L1BossBridge` contract.
- Can add and remove operators.
- Rogue owners or compromised keys may put all bridge funds at risk.

User

- Accounts that hold BBT tokens and use the `L1BossBridge` contract to deposit and withdraw them.

Operator

- Accounts approved by the bridge owner that can sign withdrawal operations.
- Rogue operators or compromised keys may put all bridge funds at risk.

[H-1] Use of Raw `create` Opcode in Assembly leads to Broken Deployment on zkSync Era

Scope `TokenFactory.sol`

Description

- Under normal behavior, the `deployToken` function allows the bridge owner to deploy a new ERC20 token contract by passing the token's bytecode.

- However, the function uses the low-level `create` opcode inside an inline assembly block. This deployment pattern is incompatible with the zkSync Era environment, where contracts must be deployed via the `ContractDeployer` system contract, and their bytecode hashes must be known to the compiler in advance. The zkSync compiler (`zksolc`) cannot process dynamically passed bytecode in this manner, which leads to broken deployments.

```
function deployToken(string memory symbol, bytes memory contractBytecode) public
onlyOwner returns (address addr) {
    assembly {
@>        addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
    }
    s_tokenToAddress[symbol] = addr;
    emit TokenDeployed(symbol, addr);
}
```

Risk

Likelihood:

- This will occur whenever the function is called on zkSync Era or any zkEVM-like chain that uses system contracts for deployment instead of native EVM opcodes.
- The `create` opcode does not fail loudly in zkSync; instead, it produces an unusable or incorrect address, leading to silent failures or misconfigured deployments.

Impact:

- Token deployments will silently fail, leaving users with no deployed token and no clear error on-chain.
- Funds or logic relying on these tokens (such as vaults or bridges) may be broken or lead to unexpected behavior due to reliance on invalid or empty token addresses.

Proof of Concept

A call to `deployToken("BBT", abi.encodePacked(L1TokenBytecode))` on zkSync will emit a `TokenDeployed` event, but the returned `addr` will be `address(0)` or a non-functional contract.

There will be no bytecode at that address, and attempts to interact with it will revert.

```
address token = factory.deployToken("BBT", L1TokenBytecode);
// token == address(0) or address with no code
require(token.code.length > 0, "Deployment failed");
```

Recommended Mitigation

To remain compatible with both Ethereum and zkSync Era, avoid using `create` in inline assembly and rely on Solidity’s `new ContractName()` syntax, which the zkSync compiler can properly handle by injecting required deployment bytecode via `factoryDeps`.

Alternatively, precompile known tokens and pass only salt or constructor params dynamically.

```diff
- assembly {
-     addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
- }
+ TokenImplementation token = new TokenImplementation(); // Pass parameters if needed
+ addr = address(token);
```

Also consider hardcoding or deploying via a minimal proxy if you're deploying many of the same token templates.

[H-2] Hardcoded Decimal Assumption can cause Incompatibility with Bridged Token Standards

Scope `L1Token.sol`

Description

- Normally, an ERC20 token uses a default of 18 decimals as implemented in OpenZeppelin's `ERC20` base contract, unless explicitly overridden.
- In the `L1Token` contract, the constructor mints `INITIAL_SUPPLY * 10 ** decimals()` tokens using the inherited `decimals()` function. However, in a bridge setup, this contract represents the L1 counterpart of tokens being moved from other chains or systems — and **not all real-world tokens use 18 decimals**. Tokens like USDT or USDC use 6 decimals. Therefore, this design falsely assumes all bridged tokens conform to 18 decimals, leading to incorrect token supply scaling.

```
constructor() ERC20("BossBridgeToken", "BBT") {
@>     _mint(msg.sender, INITIAL_SUPPLY * 10 ** decimals());
}
```

Risk

Likelihood:

- This issue will occur whenever the bridge attempts to mirror or wrap a token that does not use 18 decimals.
- Since the contract inherits from OpenZeppelin's ERC20 without overriding `decimals()`, the incorrect assumption is silently made at deployment.

Impact:

- Minted token supply will be incorrectly scaled by a factor of `10^(18 - actualDecimals)`, resulting in inflation or deflation.
- This discrepancy can cause accounting mismatches between L1 and L2, break user balances, and open potential arbitrage opportunities across chains if the supply scaling differs.

Proof of Concept

If the intention is to represent a 6-decimal stablecoin on L1:

```
INITIAL_SUPPLY = 1_000_000;
actual expected supply = 1_000_000 * 10 ** 6 = 1e12
actual minted by L1Token.sol = 1_000_000 * 10 ** 18 = 1e24
```

A user depositing 1 USDT on L2 would expect to withdraw 1 USDT on L1, but the mismatch in decimal scaling means they'll withdraw **1,000,000,000,000** tokens (or worse, withdrawals will fail due to balance inconsistencies).

Recommended Mitigation

Either:

1. **Make decimals configurable** at deployment time, so the bridge admin can specify the correct precision for the token being mirrored.
2. **Override the `decimals()` function** and explicitly set the expected value when deploying the L1Token.
3. **Avoid minting at all** in this contract and delegate supply logic to a vault or bridge manager which tracks and locks tokens externally.

```diff
- _mint(msg.sender, INITIAL_SUPPLY * 10 ** decimals());
+ uint8 customDecimals = 6; // match the L2 original token
+ _mint(msg.sender, INITIAL_SUPPLY * 10 ** customDecimals);

+ function decimals() public pure override returns (uint8) {
+     return customDecimals;
+ }
```

Or make `decimals` an immutable constructor parameter:

```
uint8 private immutable i_decimals;

constructor(uint8 decimals_) ERC20("BossBridgeToken", "BBT") {
    i_decimals = decimals_;
    _mint(msg.sender, INITIAL_SUPPLY * 10 ** decimals_);
}

function decimals() public view override returns (uint8) {
    return i_decimals;
}
```

[H-3] Arbitrary `from` Parameter Enables Token Theft via Approved Allowance

Scope `L1BossBridge.sol`

Description

- Normally, token transfers using `transferFrom(from, to, amount)` should only allow the `msg.sender` to transfer their own tokens or use a delegated allowance.
- In this case, the `depositTokensToL2` function allows any `from` address to be passed in by the caller. If a user has mistakenly approved the bridge contract for token spending, **anyone** can deposit tokens on their behalf (without their consent), transferring tokens into the vault.

```
function depositTokensToL2(address from, address l2Recipient, uint256 amount) external
whenNotPaused {
@>     token.transferFrom(from, address(vault), amount);
        emit Deposit(from, l2Recipient, amount);
}
```

Risk

Likelihood:

- This will occur whenever a user pre-approves the bridge contract and an attacker detects it.

Impact:

- Tokens can be stolen from unsuspecting users who have approved the bridge, by third-party callers.
- There is no authentication to restrict the usage of the `from` address.

Proof of Concept

1. Alice approves the bridge to spend 100 tokens: `token.approve(bridge, 100)`
2. Malicious actor calls: `bridge.depositTokensToL2(alice, maliciousL2Addr, 100)`
3. Alice’s tokens are moved to the vault and credited to someone else on L2.

Recommended Mitigation

Ensure that only `msg.sender` can transfer their tokens to the vault:

```diff
- function depositTokensToL2(address from, address l2Recipient, uint256 amount)
external whenNotPaused {
-     token.transferFrom(from, address(vault), amount);
+ function depositTokensToL2(address l2Recipient, uint256 amount) external
whenNotPaused {
+     token.transferFrom(msg.sender, address(vault), amount);
      emit Deposit(msg.sender, l2Recipient, amount);
}
```

[H-4] Vault Can Be Used as the `from` Address to Drain It

Scope `L1BossBridge.sol`

Description

- The same `depositTokensToL2` function allows the caller to specify **any `from` address**, including the **vault address itself**.
- If the vault has received token approvals (unlikely but possible if reused or upgraded), someone could use `vault` as the `from` and pull tokens **out of the vault**, not into it.

```
function depositTokensToL2(address from, address l2Recipient, uint256 amount) external
whenNotPaused {
@>     token.transferFrom(from, address(vault), amount);
}
```

Risk

Likelihood:

- This can happen if the vault is reused, upgraded, or misconfigured to approve the token or has a balance or allowance.

Impact:

- Vault funds can be transferred by any user calling this function, draining L1 bridge reserves.

Proof of Concept

If the vault for any reason has an allowance set (manually or via prior interaction):

```
token.approve(bridge, type(uint256).max)  // done internally already

// Any user:
bridge.depositTokensToL2(address(vault), attackerL2, 100);
```

Recommended Mitigation

Same as the previous fix — only allow `msg.sender` as the source of funds:

```diff
- function depositTokensToL2(address from, address l2Recipient, uint256 amount)
external whenNotPaused {
-     token.transferFrom(from, address(vault), amount);
+ function depositTokensToL2(address l2Recipient, uint256 amount) external
whenNotPaused {
+     token.transferFrom(msg.sender, address(vault), amount);
}
```

[H-5] No Replay Protection on Withdraw Signatures

Scope `L1BossBridge.sol`

Description

- The `withdrawTokensToL1` function uses a signed message to authorize withdrawals. However, there is **no nonce or unique replay protection**, meaning the same signature can be reused to withdraw repeatedly.

```
function withdrawTokensToL1(address to, uint256 amount, uint8 v, bytes32 r, bytes32 s)
external {
@>     sendToL1(
            v,
            r,
            s,
            abi.encode(
                address(token),
                0,
                abi.encodeCall(IERC20.transferFrom, (address(vault), to, amount))
            )
        );
}
```

Risk

Likelihood:

- This will occur whenever a user reuses a signature or an attacker obtains one. The bridge will not prevent re-execution.

Impact:

- Attackers can **drain the vault** using a valid signature multiple times. One valid withdrawal signature can be exploited repeatedly.

Proof of Concept

1. Bob signs a valid withdrawal message: `ECDSA.sign(keccak256(...), BobPrivateKey)`
2. The user calls `withdrawTokensToL1(...)` → Success
3. Attacker captures the calldata and replays it again → Success again

Recommended Mitigation

Introduce nonce-based replay protection. For example:

```
mapping(bytes32 => bool) public usedMessages;

function sendToL1(...) public {
    bytes32 msgHash = keccak256(message);

    require(!usedMessages[msgHash], "Replay detected");
    usedMessages[msgHash] = true;

    address signer = ECDSA.recover(...);
    ...
}
```

Alternatively, enforce **unique withdrawal IDs** or include a per-user nonce in the message hash.

```diff
- sendToL1(v, r, s, abi.encode(...));
+ bytes32 uniqueId = keccak256(abi.encode(..., nonce));
+ require(!usedMessages[uniqueId], "Replay");
+ usedMessages[uniqueId] = true;
```
