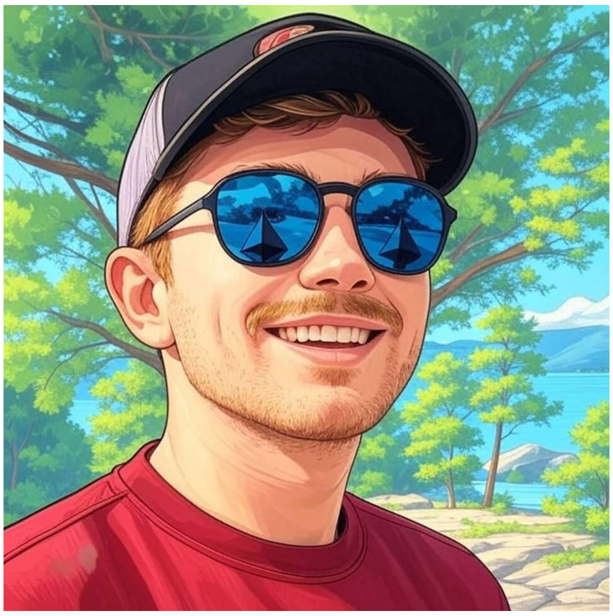

Boss Bridge Audit Report

Version 1.0

August 19, 2025

Boss Bridge Audit Report

Mustby.eth

August 19, 2025

Prepared by: Mustby.eth Lead Auditors: - Matthew Ustby

Table of Contents

  - Low
    * [L-1] Lack of event emissions for withdrawals and sending tokens to the L1.
  - Informational
    * [I-1] `DEPOSIT_LIMIT` should be constant in `L1BossBridge.sol`.
    * [I-2] Need to follow CEI in `L1BossBridge::depositTokensToL2`.
    * [I-3] Variable should be immutable in `L1Vault.sol`.
    * [I-4] `L1Vault::approveTo` should check the return value of approve.
    * [I-5] Insufficient overall test coverage.
  - Gas

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 mustby.eth 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
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Impact

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

The audit went well. We found a total of 11 vulnerabilities.

Issues found

Severity	Number of Issues Found
High	4
Medium	1
Low	1
Info	5
Gas	0
Total	11

Findings

High

[H-1] Arbitrary transfer from allows any user can steal funds after approval.

Description: 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).

Impact: User can steal eachothers’ funds!

Proof of Concept: Insert this test into the `L1BossBridge.t.sol` file:

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

        // Bob
        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);

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

Recommended Mitigation: 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);
}
```

[H-2] Users can mint unlimited tokens to the L2.

Description: The `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: Unlimited unbacked tokens can be minted on L2, leading to theft of funds or inflation on the L2 side.

Proof of Concept: Insert this test into the `L1TokenBridge.t.sol` file:

```
    function testCanTransferFromVaultToAttacker() public {
        address attacker = makeAddr("attacker");

        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance);

        // Can trigger the deposit event, self transfering tokens to
            the vault
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);

        // Can do this forever?
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
    }
```

This test shows that the attacker and mint and steal unlimited tokens.

Recommended Mitigation: Remove the infinite approval from the L1Vault constructor, as it creates an unnecessary attack surface given the vault's ownership by the bridge. Instead, implement a push-based transfer mechanism in L1Vault for withdrawals:

```
    // In L1Vault.sol
    function sendTokens(address _to, uint256 _amount) external
        onlyOwner {
    token.transfer(_to, _amount);
    }
```

Update the withdrawal function in `L1BossBridge` to call `vault.sendTokens(_to, _amount)` instead of `token.transferFrom(address(vault), _to, _amount)`. This maintains withdrawal functionality while preventing abuse of approvals in deposits.

[H-3] Signatures can be reused in `L1BossBridge::sendToL1`.

Description: The sendToL1 function in L1BossBridge.sol does not include a mechanism to prevent signature reuse. An attacker can replay a valid signature multiple times to execute unauthorized withdrawals, as the function does not validate whether a signature has been used before.

Impact: An attacker can repeatedly use a single valid signature to drain tokens from the vault, leading to significant financial loss. The proof of concept demonstrates that an attacker can steal the entire vault balance by replaying a signed withdrawal message, undermining the bridge's security.

Proof of Concept: Insert this test into `L1TokenBridge.t.sol`:

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

        // An attacker deposits tokens to L2
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(attacker, attacker,
            attackerInitialBalance);

        // Signer/Operator is going to sign the 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)));

        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);
    }
```

This test will pass, showing that the attacker was able to successfully reuse the signature and steal the balance from the vault address.

Recommended Mitigation: Add a nonce to the signed message and track used nonces in L1BossBridge.sol to ensure signatures are used only once. Update the sendToL1 function to validate the nonce and mark it as used.

```
    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
        message) public nonReentrant whenNotPaused {
        if (message.length > MAX_DATA_SIZE) {
            revert L1BossBridge__DataTooLarge();
        }

        address signer = ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(message)), v, r, s);
        if (!signers[signer]) {
            revert L1BossBridge__Unauthorized();
        }

+         // Decode message with nonce
+         (address target, uint256 value, bytes memory data, uint256 nonce) = abi.decode(message, (address, uint256, bytes, uint256));

+         // Check and mark nonce as used
+         if (usedNonces[signer][nonce]) {
+             revert L1BossBridge__NonceAlreadyUsed();
+         }
+         usedNonces[signer][nonce] = true;

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

        emit WithdrawalSent(target, value, data, signer);
    }
```

In addition, add this new nonce tracker and error message

```
+    mapping(address => mapping(uint256 => bool)) public usedNonces;
.
.
.
+    error L1BossBridge__NonceAlreadyUsed();
```

[H-4] CREATE Opcode in `TokenFactory::deployToken` does not work on ZKSync.

Description: The `deployToken` function in `TokenFactory.sol` uses the CREATE opcode in assembly to deploy a new contract. This opcode is not supported on ZKSync Era, which uses a different contract deployment mechanism due to its zkEVM architecture.

```
    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);
        }
```

Impact: The deployToken function will fail on ZKSync Era, preventing token deployment and breaking core functionality of the bridge on this chain.

Proof of Concept: On ZKSync Era, the CREATE opcode is not supported because the zkEVM uses a custom deployment mechanism involving system contracts. Attempting to execute:

```
        addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
```

will revert or fail silently, resulting in addr being 0x0, and no token contract will be deployed.

Recommended Mitigation: Replace the CREATE opcode with ZKSync's CREATE2 factory contract (ContractDeployer) to deploy tokens. Use the IContractDeployer interface to ensure compatibility.

Medium

[M-1] Gas bomb in `L1BossBridge::sendToL1`.

Description: In `L1BossBridge::sendToL1`, we need to mitigate the potential for excessive gas consumption in the sendToL1 function, which uses a low-level call with arbitrary 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));

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

Recommended Mitigation: Modify the sendToL1 function to impose a gas limit on the low-level call to ensure it cannot consume excessive gas. Additionally, consider adding input validation to restrict the size of the data parameter, as large data payloads can contribute to high gas costs.

```
    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public nonReentrant whenNotPaused {
+       // Validate data size to prevent gas bomb
+       if (message.lenge > MAX_DATA_SIZE) {
+           revert L1BossBridge__DataTooLarge();
        }

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

Here are the three additional variables to add above:

```
+   // Gas limit for low-level calls
+   uint256 private constant CALL_GAS_LIMIT = 100_000;

+   // Maximum data size to prevent oversized payloads (e.g., enough for transferFrom encoding)
+   uint256 private constant MAX_DATA_SIZE = 100;
.
.
.
+   error L1BossBridge__DataTooLarge();
```

Low

[L-1] Lack of event emissions for withdrawals and sending tokens to the L1.

Description: Neither the sendToL1 function nor the 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.

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

```
+   event WithdrawalSent(address indexed target, uint256 value, bytes
   data, address indexed signer);
```

```
    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));

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

+       // Emit event after successful call
+       emit WithdrawalSent(target, value, data, signer);
    }
```

The event adds minimal gas overhead and enhances transparency.

Informational

[I-1] `DEPOSIT_LIMIT` should be constant in `L1BossBridge.sol`.

Description:

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

Recommended Mitigation:

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

Adding constant forces the variable to be hardcoded at compile time. This aligns with the README that mentions “magic numbers defined as literals should be constants.”

[I-2] Need to follow CEI in `L1BossBridge::depositTokensToL2`.

Description: The current `depositeTokensToL2` function does not follow CEI.

```
    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);
@>      emit Deposit(from, l2Recipient, amount);
    }
```

Recommended Mitigation:

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

This fix now follows CEI: Checks, Effects, Interactions. Even though there is no reentrancy vulnerability here, this is the best practice.

[I-3] Variable should be immutable in `L1Vault.sol`.

Description:

```
    IERC20 public token;
```

Recommended Mitigation:

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

The immutable keyword ensures that a state variable's value is fixed at construction time and cannot be changed. This reduces gas costs and enhances security by preventing unintended modifications.

[I-4] `L1Vault::approveTo` should check the return value of approve.

Description: The function `approveTo` in `L1Vault.sol` should be checking the return value of `token.approve`.

```
    function approveTo(address target, uint256 amount) external
       onlyOwner {
```

```
@>        token.approve(target, amount);
    }
```

Recommended Mitigation:

```diff
    function approveTo(address target, uint256 amount) external onlyOwner {
-        token.approve(target, amount);
+        require(token.approve(target, amount), "L1Vault: Approval failed");
    }
```

[I-5] Insufficient overall test coverage.

```
| File                 | % Lines        | % Statements   | % Branches     | % Funcs        |
| -------------------- | -------------- | -------------- | -------------- | -------------- |
| src/L1BossBridge.sol | 86.67% (13/15) | 90.00% (18/20) | 83.33% (5/6)   | 83.33% (5/6)   |
| src/L1Vault.sol      | 0.00% (0/1)    | 0.00% (0/1)    | 100.00% (0/0)  | 0.00% (0/1)    |
| src/TokenFactory.sol | 100.00% (4/4)  | 100.00% (4/4)  | 100.00% (0/0)  | 100.00% (2/2)  |
| Total                | 85.00% (17/20) | 88.00% (22/25) | 83.33% (5/6)   | 77.78% (7/9)   |
```

Recommended Mitigation: Aim to get test coverage above 90% for all files.

Gas