BossBridge Audit Report

Version 1.0

Cyfrin.io

June 13, 2024

Protocol Audit Report

Seven Cedars

June 10, 2024

Prepared by: Seven Cedars Lead Auditors: Seven Cedars

Table of Contents

* [H-3] The `L1BossBridge::sendToL1` is vulnerable to replay signature attacks, because it lacks a check if a signature has already been used. It means that a signature can be used an unlimited times to retrieve L1 tokens.
* [H-4] The `L1BossBridge::sendToL1` function does not check the values extracted from `message` before using them to send eth to recipient address by placing a low level `.call`. This allows a malicious user to sent large amounts of eth to their own address, or create a DoS by executing a function with immense gas cost.

Protocol Summary

This project presents a simple bridge mechanism to move our ERC20 token from L1 to an L2 we're building. The L2 part of the bridge is not included.

The bridge 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.

To ensure user safety, this first version of the bridge has a few security mechanisms in place:

The developers plan on launching `L1BossBridge` on both Ethereum Mainnet and ZKSync.

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

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)

Actors/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

I only focused on high risk vulnerabilities for this report.

Severity	Number of Issues found
high	4
medium	0
low	0
total	4

Issues found

Findings

High

[H-1] In the `L1BossBridge::depositTokensToL2` function an arbitrary `from` address is passed into `safeTransferFrom`. It allows a malicious user to transfer tokens that have been approved by another user to the vault, setting `l2Recipient` in the procces and stealing the related L2 tokens.

Description: The `depositTokensToL2` is meant to allow a user to deposit L1 tokens to the L1Vault and receive L2 tokens in return. To do this, the function takes a `from` field (the user sending the funds), a `l2Recipient` address (the user receiving the L2 tokens) and an `amount` value (the humber of tokens to send and receive).

However, because the `from` address can be set to any address, it is possible for a malicious user Bob to send tokens that have previously been approved for transfer by benevolent user Alice. Doing so, Bob can set the `l2Recipient` address to his own address and send the maximum approved amount of tokens as `amount`.

```
1 @>  function depositTokensToL2(address from, address l2Recipient,
      uint256 amount) external whenNotPaused {
2         if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
3             revert L1BossBridge__DepositLimitReached();
4         }
5         token.safeTransferFrom(from, address(vault), amount);
6
```

```
7  @>      emit Deposit(from, l2Recipient, amount);
8      }
```

Impact: When a malicious user calls the `depositTokensToL2` function with someone elses tokens, it results in a `Deposit` event being emitted that will tell the bridge to transfer the equivalent amount of L2 tokens to the malicious user. In effect, the malicious user is stealing all L2 tokens from the benevolent user.

Proof of Concept: 1. Benevolent user Alice approves tokens. 2. Malicious user Bob depositTokensToL2 Alice's tokens to `depositTokensToL2` while setting `l2Recipient` to his own address and `amount` to all tokens Alice holds. 3. An event is emitted that has Bob's address as recipient, with all of Alice's tokens now in the L1 vault.

Proof of Concept

Place the following in `L1TokenBridge.t.sol`

```
1      function testCanMoveApprovedTokensOfOtherUsers() public {
2          // Alice
3          vm.startPrank(user);
4          token.approve(address(tokenBridge), type(uint256).max);
5
6          // Bob
7          uint256 depositAmount = token.balanceOf(user);
8          address attacker = makeAddr("attacker");
9          vm.startPrank(attacker);
10         vm.expectEmit(address(tokenBridge));
11         emit Deposit(user, attacker, depositAmount);
12         tokenBridge.depositTokensToL2(user, attacker, depositAmount);
13
14         assertEq(token.balanceOf(user), 0);
15         assertEq(token.balanceOf(address(vault)), depositAmount);
16         vm.stopPrank();
17     }
```

Recommended Mitigation: Do not pass an arbitrary value into the `from` address. It is better to use `msg.sender` as a value.

```
1  +  function depositTokensToL2(address l2Recipient, uint256 amount)
       external whenNotPaused {
2  -  function depositTokensToL2(address from, address l2Recipient,
       uint256 amount) external whenNotPaused {
3          if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
4              revert L1BossBridge__DepositLimitReached();
5          }
6  +        token.safeTransferFrom(msg.sender, address(vault), amount);
7  -        token.safeTransferFrom(from, address(vault), amount);
8
9  +      emit Deposit(msg.sender, l2Recipient, amount);
```

```
10 -       emit Deposit(from, l2Recipient, amount);
11     }
```

[H-2] By setting `from` in the `L1BossBridge::depositTokensToL2` function as `address(vault)` and `l2Recipient` as the attacker address, it is possible to mint an almost unlimited amount of L2 tokens.

Description: As noted above, the `depositTokensToL2` is meant to allow a user to deposit L1 tokens to the L1Vault and receive L2 tokens in return. To do this, the function takes a `from` field (the user sending the funds), a `l2Recipient` address (the user receiving the L2 tokens) and an `amount` value (the humber of tokens to send and receive).

However, in addition to the `from` field taking an arbitrary value (see the issue [H-1] above):

First, the L1 vault is given full approval over all its tokens in its vault at time of construction of the vault:

```
1      constructor(IERC20 _token) Ownable(msg.sender) {
2          token = _token;
3          vault = new L1Vault(token);
4  @>      vault.approveTo(address(this), type(uint256).max);
5      }
```

Second, the documentation notes that "Successful deposits trigger an event that our off-chain mechanism picks up, parses it and *mints the corresponding tokens on L2*".

Impact: Together, the above issues allow a malicious user to enter `address(vault)` as the from address, and have it send the full amount deposited in the vault to itself, trigger a `Deposit` event and have the corresponding amount of L2 tokens minted and send to their address. A malicious user can repeat this process indefinitely.

Proof of Concept: 1. Malicious user calls the `depositTokensToL2` with the vaults full balance and the attacker's address as `l2Recipient`. 2. The transaction passes. 3. A `Deposit` event is emitted, with attacker's address as recipient.

Proof of Concept

Place the following in `L1TokenBridge.t.sol`

```
1      function testCanTransferFromVaultToVault() public {
2          address attacker = makeAddr("attacker");
3          uint256 vaultBalance = 500 ether;
4          deal(address(token), address(vault), vaultBalance);
5
6          // the following should trigger deposit event.
7          vm.expectEmit(address(tokenBridge));
```

```
8           emit Deposit(address(vault), attacker, vaultBalance);
9           tokenBridge.depositTokensToL2(address(vault), attacker,
                vaultBalance);
10      }
```

Recommended Mitigation: There are several mitigation that can be implemented. 1. Do not pass an arbitrary value to the `from` field. See also vulnerability [H-1] above. 2. Disallow `address(vault)` to deposit tokens altogether. In addition to the changes proposed above:

```
1     function depositTokensToL2(address l2Recipient, uint256 amount)
          external whenNotPaused {
2           if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
3               revert L1BossBridge__DepositLimitReached();
4           }
5   +        if (msg.sender == address(vault)) {
6   +            revert L1BossBridge__VaultCannotDeposit();
7   +        }
8
9           token.safeTransferFrom(msg.sender, address(vault), amount);
10
11        emit Deposit(msg.sender, l2Recipient, amount);
12      }
```

3. It is also possible to restrict transfer approvals, but this will trigger a broader refactoring of the code base.

[H-3] The `L1BossBridge::sendToL1` is vulnerable to replay signature attacks, because it lacks a check if a signature has already been used. It means that a signature can be used an unlimited times to retrieve L1 tokens.

Description: The `sendToL1` function is meant to allow a central signer to approve and execute retrievals of L1 tokens following the deposit of L2 tokens. To do this, it takes the `v`, `r` and `s` values as signature of the `L1BossBridge` signer, and a `message` field that contains the abi.encoded function call to transfer L1 tokens to the user.

However, the `sendToL1` function lacks a check if the signature has been used before. Because the signature is send over chain, it can be copied by a malicious user. This user can use these values to authorize subsequent calls to the `sendToL1` function indefinitely.

Impact: As a malicious user can use a legitimate signature to withdraw L1 tokens indefinitely, it allows all assets to be drained from the vault.

Proof of Concept: 1. A malicious user makes a legit deposit to `depositTokensToL2`. 2. This triggers the operator of the vault to sign a message for the function `withdrawTokensToL1`. 3. The malicious

user copies the resulting signature and replays the call to `withdrawTokensToL1` until all funds are drained.

Proof of Concept

Place the following in `L1TokenBridge.t.sol`

```
    function testSignatureReplay() public {
        address attacker = makeAddr("attacker");
        uint256 vaultInitialBalance = 100e18;
        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, attacker, attackerInitialBalance);

        //signer/operator signs withdrawal
        bytes memory message = abi.encode(
            address(token), 0, abi.encodeCall(IERC20.transferFrom, (address(vault), attacker, attackerInitialBalance))
        );
        (uint8 v, bytes32 r, bytes32 s) =
            vm.sign(operator.key, MessageHashUtils.toEthSignedMessageHash(keccak256(message)));

        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: Add a state variable to keep track what signatures have been used, and adding a check to disallow reuse of signatures.

```
+   mapping(bytes signature => bool hasBeenUsed) public usedSignatures; // users that can send l1 -> l2
.
.
.
+   error L1BossBridge__ReuseSignatureNotAllowed();
.
.
```

```
8  .
9      function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
           message) public nonReentrant whenNotPaused {
10         bytes memory signature = abi.encodePacked(r, s, v)
11         address signer = ECDSA.recover(MessageHashUtils.
               toEthSignedMessageHash(keccak256(message)), v, r, s);
12
13         if (!signers[signer]) {
14             revert L1BossBridge__Unauthorized();
15         }
16
17 +       if (usedSignatures[signature]) {
18 +           revert L1BossBridge__ReuseSignatureNotAllowed();
19 +       }
20 +       usedSignatures[signature] = true;
21
22
23         (address target, uint256 value, bytes memory data) = abi.decode
               (message, (address, uint256, bytes));
24         (bool success,) = target.call{ value: value }(data);
25         if (!success) {
26             revert L1BossBridge__CallFailed();
27         }
28     }
```

[H-4] The `L1BossBridge::sendToL1` function does not check the values extracted from `message` before using them to send eth to recipient address by placing a low level `.call`. This allows a malicious user to sent large amounts of eth to their own address, or create a DoS by executing a function with immense gas cost.

Description: As mentioned above, the `sendToL1` function is meant to allow a central signer to approve and execute retrievals of L1 tokens following the deposit of L2 tokens. To do this, it takes the `v`, `r` and `s` values as signature of the `L1BossBridge` signer, and a `message` field that contains the abi.encoded function call to transfer L1 tokens to the user.

The function itself encodes the message using the `v`, `r` and `s` values and subsequently retrieves the address from the encrypted message. This all happens within one line:

```
1      address signer = ECDSA.recover(MessageHashUtils.
           toEthSignedMessageHash(keccak256(message)), v, r, s);
```

However, because the message and signature are sent in separate files, it is possible to change the `message` value to anything.

Impact: In combination with front running the transaction, it allows a malicious user to send any value to their own address.

Recommended Mitigation: Send the signature as a signed message. This will mage it impossible to change the content of the message, as it will alter the address that is retrieved.

```
+
        // note: `signature` should be the result of hashing the
            message and v, r, s through an external function.
+       function sendToL1(bytes32 signature, bytes memory message)
    public nonReentrant whenNotPaused {
-       function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
    message) public nonReentrant whenNotPaused {
        bytes memory signature = abi.encodePacked(r, s, v);
-       address signer = ECDSA.recover(MessageHashUtils.
    toEthSignedMessageHash(keccak256(message)), v, r, s);
+       bytes32 digest = MessageHashUtils.toEthSignedMessageHash(
    keccak256(message));
+       address signer = digest.recover(signature);

        if (!signers[signer]) {
            revert L1BossBridge__Unauthorized();
        }

        (address target, uint256 value, bytes memory data) = abi.decode
            (message, (address, uint256, bytes));
```