BossBridge Audit Report

Version 1.0

https://github.com/Heavens01

August 16, 2025

BossBridge Audit Report

Heavens01

August 15, 2025

Prepared by: Heavens01 Lead Security Researcher(s): - Heavens01

Table of Contents

    * Description:
    * Impact:
    * Proof of Concept:
    * Recommended Mitigation:
  - [H-2] Malicious User Can Steal Tokens by Specifying Any Approved from Address (Lack of Caller Restriction + Token Theft)
    * Description:
    * Impact:
    * Proof of Concept:
    * Recommended Mitigation:
  - [H-3] Denial of Service by Bypassing Deposit Limit (Improper Limit Enforcement + Denial Of Service)
    * Description:
    * Impact:
    * Proof of Concept:
    * Recommended Mitigation:
  - [H-4] Signature Replay Attack in Withdrawal Function (Lack of Nonce and Expiration + Unauthorized Token Withdrawal)
    * Description:
    * Impact:
    * Proof of Concept:
    * Recommended Mitigation:
  - [H-5] TokenFactory Mints L1Token Supply to Deployer (Improper Token Allocation + Permanent Lockup)
    * Description:
    * Impact:
    * Proof of Concept:
    * Recommended Mitigation:
  - [H-6] Use of Era-VM Incompatible Opcode in TokenFactory (Non-Compatible Assembly + Deployment Failure on ZKSync)
    * Description:
    * Impact:
    * Proof of Concept:
    * Recommended Mitigation:
  - MEDIUM
  - [M-1] Malicious Data in sendToL1 Can Approve Vault Tokens to Attacker (Unrestricted Call Data + Vault Drainage)

    * Description:
    * Impact:
    * Proof of Concept:
    * Recommended Mitigation:
  - [M-2] Malicious Data in sendToL1 Can Cause Excessive Gas Consumption (Unrestricted Call Data + Gas Griefing)
    * Description:
    * Impact:
    * Proof of Concept:
    * Recommended Mitigation:
  - [L-1] Multiple Tokens allows Same Symbol Overwrite Previous Mapping (Lack of Symbol Uniqueness + Loss of Token Reference)
    * Description:
    * Impact:
    * Proof of Concept:
    * Recommended Mitigation:
  - [L-2] Lack of Event Emission in sendToL1 and withdrawTokensToL1 (Missing Event Logging + Reduced Transparency)
    * Description:
    * Impact:
    * Proof of Concept:
    * Recommended Mitigation:
  - [L-3] Signer Disablement Causes Transaction Failure (Signer Status Change + Transaction Disruption)
    * Description:
    * Impact:
    * Proof of Concept:
    * Recommended Mitigation:

Protocol Summary

Boss Bridge

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.

Project Link

```
git clone https://github.com/Cyfrin/7-boss-bridge-audit
cd 7-boss-bridge-audit
make
```

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 HEAVENS01 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 Scope 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

*We spent 2 Days (48 Hours) with one auditor using solidity metrics tool, aderyn, slither, and manual reading of codebase.

Issues found

Severity	Number of issues found
High	6
Medium	2
Low	3
Info/Gas	Not Included
Total	13

Findings

High

[H-1] Unrestricted Deposit Allows Infinite L2 Minting (Lack of Caller Restriction + Unauthorized Token Minting)

Description:

The `depositTokensToL2` function in `L1BossBridge.sol` allows any user to call the function with any `from` address that has approved the contract, enabling a malicious user to trigger the `Deposit` event repeatedly with the vault's address as `from` (and this is because the `L1BossBridge` has been approved of `type(uint256).max` token of `L1Vault`). This causes the off-chain service to mint corresponding tokens on L2 without transferring tokens from the caller, effectively allowing infinite minting on L2.

Impact:

A malicious user can exploit this vulnerability to mint an unlimited amount of tokens on L2 by repeatedly emitting `Deposit` events, specifying the vault's address as `from`. This bypasses the intended token transfer mechanism, undermining the bridge's integrity and potentially flooding the L2 with unauthorized tokens, leading to economic imbalance and loss of trust in the system.

Proof of Concept:

The issue is noted in `L1BossBridge.sol`:

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

Appending this test in `L1BossBridgeTest.sol` demonstrates the issue:

Test

```
    function
        testMaliciousUserCanMintInfiniteAmountOnL2PosingVaultAsFrom()
        public {
        // Deals contract some token (that same token)
        uint256 vaultBalance = 1000 ether;
        deal(address(token), address(vault), vaultBalance);

        // Malicious User steals user token by first calling
            depositTokensToL2 with user's address as from
        address maliciousUser = makeAddr("maliciousUser");
        vm.prank(maliciousUser);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), maliciousUser, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), maliciousUser,
            vaultBalance);

        // Malicious User can do it again and again and keep minting on
             L2
        vm.prank(maliciousUser);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), maliciousUser, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), maliciousUser,
            vaultBalance);
    }
```

Recommended Mitigation:

Restrict `depositTokensToL2` to only allow the caller to deposit their own tokens by enforcing `from == msg.sender`:

```
contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
    function depositTokensToL2(address from, address l2Recipient,
        uint256 amount) external whenNotPaused {
+       if (from != msg.sender) {
+           revert L1BossBridge__Unauthorized();
+       }
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
        token.safeTransferFrom(from, address(vault), amount);
        emit Deposit(from, l2Recipient, amount);
    }
}
```

[H-2] Malicious User Can Steal Tokens by Specifying Any Approved from Address (Lack of Caller Restriction + Token Theft)

Description:

The `depositTokensToL2` function in `L1BossBridge.sol` allows any user to specify any `from` address that has approved the contract, enabling a malicious user to transfer tokens from an innocent user's address to the vault and emit a `Deposit` event crediting the malicious user's L2 address. This allows token theft from users who have approved the contract.

Impact:

A malicious user can drain tokens from any account that has approved the `L1BossBridge` contract, transferring them to the `vault` and receiving equivalent tokens on L2. This undermines user trust, leads to loss of funds for innocent users, and compromises the bridge's security.

Proof of Concept:

Appeding this test in `L1BossBridgeTest.sol` demonstrates this issue:

```
    function testMaliciousUserStealsFromUsersWhoApprovesContract()
        public {
        // User approves
        vm.prank(user);
        uint256 amount = 1000e18;
        token.approve(address(tokenBridge), amount);
        address maliciousUser = makeAddr("maliciousUser");

        // Malicious User steals user token by first calling
           depositTokensToL2 with user's address as from
        vm.prank(maliciousUser);
        vm.expectEmit(address(tokenBridge));
        emit Deposit(user, maliciousUser, amount);
        tokenBridge.depositTokensToL2(user, maliciousUser, amount);

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

Recommended Mitigation:

Restrict `depositTokensToL2` to only allow the caller to deposit their own tokens by enforcing `from` == `msg.sender`:

```
contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
    function depositTokensToL2(address from, address l2Recipient,
        uint256 amount) external whenNotPaused {
+        if (from != msg.sender) {
+            revert L1BossBridge__Unauthorized();
+        }
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
        token.safeTransferFrom(from, address(vault), amount);
        emit Deposit(from, l2Recipient, amount);
    }
}
```

[H-3] Denial of Service by Bypassing Deposit Limit (Improper Limit Enforcement + Denial Of Service)

Description:

The `depositTokensToL2` function in `L1BossBridge.sol` checks the vault's token balance against `DEPOSIT_LIMIT` to prevent excessive deposits. However, tokens can be sent directly to the vault via `transfer` or `transferFrom`, bypassing this check. This inflates the vault's balance, causing legitimate deposits to revert with `L1BossBridge__DepositLimitReached`, enabling a denial of service (DoS) attack.

Impact:

A malicious user can send tokens directly to the vault, pushing its balance over `DEPOSIT_LIMIT`, which blocks all users from depositing tokens via `depositTokensToL2`. This disrupts the bridge's functionality, preventing users from transferring tokens to L2 and undermining the bridge's reliability.

Proof of Concept:

The issue is noted in `L1BossBridge.sol`:

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

```
    }
```

Appending this test in `L1BossBridgeTest.sol` demonstrates the issue:

```
function testDoSByDirectVaultTransfer() public {
    address maliciousUser = makeAddr("maliciousUser");
    uint256 amount = tokenBridge.DEPOSIT_LIMIT();
    deal(address(token), maliciousUser, amount);

    vm.startPrank(maliciousUser);
    token.approve(address(vault), amount);
    token.transfer(address(vault), amount);

    vm.startPrank(user);
    token.approve(address(tokenBridge), 10e18);
    vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
        selector);
    tokenBridge.depositTokensToL2(user, userInL2, 10e18);
    vm.stopPrank();
}
```

Recommended Mitigation:

Track deposits using a mapping to record the total deposited amount instead of relying on the vault's balance:

```
contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
+   mapping(address => uint256) public depositedAmounts;
+   uint256 public totalDeposited;
    uint256 public DEPOSIT_LIMIT = 100_000 ether;

    function depositTokensToL2(address from, address l2Recipient,
        uint256 amount) external whenNotPaused {
-       if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
+       if (totalDeposited + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
+       depositedAmounts[from] += amount;
+       totalDeposited += amount;
        token.safeTransferFrom(from, address(vault), amount);
        emit Deposit(from, l2Recipient, amount);
    }
}
```

[H-4] Signature Replay Attack in Withdrawal Function (Lack of Nonce and Expiration + Unauthorized Token Withdrawal)

Description:

The `sendToL1` function in `L1BossBridge.sol` verifies signatures to authorize withdrawals but lacks nonce and expiration checks. This allows a malicious user to reuse a valid signature multiple times, repeatedly withdrawing tokens from the vault without additional authorization, draining the vault's funds.

Impact:

A malicious user can replay a signed withdrawal message to extract tokens from the vault multiple times, leading to unauthorized token withdrawals. This can drain the vault, causing significant financial loss and undermining the bridge's security and trust.

Proof of Concept:

Appending this test in `L1BossBridgeTest.sol` demonstrates this issue:

```
function testSignatureReplayAttack() public {
    address maliciousUser = makeAddr("maliciousUser");
    uint256 malUserInitialBal = 1000 ether;
    deal(address(token), maliciousUser, malUserInitialBal);
    uint256 initialVaultBalance = 1000 ether;
    deal(address(token), address(vault), initialVaultBalance);

    vm.startPrank(maliciousUser);
    token.approve(address(tokenBridge), malUserInitialBal);
    tokenBridge.depositTokensToL2(maliciousUser, maliciousUser,
        malUserInitialBal);

    bytes memory message = abi.encode(
        address(token),
        0,
        abi.encodeCall(IERC20.transferFrom, (address(vault),
            maliciousUser, malUserInitialBal))
    );

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

    while (token.balanceOf(address(vault)) > 0) {
```

```
        tokenBridge.withdrawTokensToL1(maliciousUser, malUserInitialBal
            , v, r, s);
    }

    assertEq(token.balanceOf(address(vault)), 0);
    assertEq(token.balanceOf(maliciousUser), malUserInitialBal +
        initialVaultBalance);
}
```

Recommended Mitigation:

Add a nonce and expiration timestamp to the signed message, and track used nonces to prevent replay attacks:

```
contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
+   mapping(address => uint256) public nonces;
+   mapping(bytes32 => bool) public usedSignatures;

    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
        message) public nonReentrant whenNotPaused {
+       bytes32 messageHash = keccak256(message);
+       if (usedSignatures[messageHash]) {
+           revert L1BossBridge__Unauthorized();
+       }
+       usedSignatures[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, uint256
            nonce, uint256 expiration) =
+           abi.decode(message, (address, uint256, bytes, uint256,
    uint256));
+       if (nonce != nonces[signer] || block.timestamp > expiration) {
+           revert L1BossBridge__Unauthorized();
+       }
+       nonces[signer]++;
        (bool success,) = target.call{ value: value }(data);
        if (!success) {
            revert L1BossBridge__CallFailed();
        }
    }

    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)),
+                   nonces[msg.sender],
+                   block.timestamp + 1 hours
                )
            );
        }
}
```

[H-5] TokenFactory Mints L1Token Supply to Deployer (Improper Token Allocation + Permanent Lockup)

Description:

The `deployToken` function in `TokenFactory.sol` deploys new `L1Token` contracts, which mint their entire initial supply (1,000,000 tokens) to the deployer (`msg.sender` of `L1Token`). If the deployer is the `TokenFactory` contract itself, these tokens are locked in the factory, as it lacks a mechanism to transfer or distribute them, rendering the tokens inaccessible.

Impact:

The initial token supply of each deployed `L1Token` is minted to the `TokenFactory` contract, which cannot distribute or use them. This locks the tokens forever, preventing their use in the bridge or by users, disrupting the token ecosystem and causing economic loss or rendering the deployed tokens useless.

Proof of Concept:

In `TokenFactory.sol`:

```
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);
}
```

In `L1Token.sol`:

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

Appending this test in `L1BossBridgeTest.sol` demonstrates the issue:

```
function testTokenFactoryMintsToItself() public {
    vm.startPrank(deployer);
    TokenFactory factory = new TokenFactory();
    bytes memory bytecode = type(L1Token).creationCode;
    address tokenAddr = factory.deployToken("BBT", bytecode);
    L1Token newToken = L1Token(tokenAddr);
    assertEq(newToken.balanceOf(address(factory)), 1_000_000 * 10 ** newToken.decimals());
    assertEq(newToken.totalSupply(), 1_000_000 * 10 ** newToken.decimals());
    vm.stopPrank();
}
```

Recommended Mitigation:

Modify `L1Token` to accept a recipient address for minting and pass a designated address (e.g., the bridge) from `TokenFactory`:

```
// In L1Token.sol
contract L1Token is ERC20 {
    uint256 private constant INITIAL_SUPPLY = 1_000_000;

+   constructor(address initialRecipient) ERC20("BossBridgeToken", "BBT") {
-   constructor() ERC20("BossBridgeToken", "BBT") {
-       _mint(msg.sender, INITIAL_SUPPLY * 10 ** decimals());
+       _mint(initialRecipient, INITIAL_SUPPLY * 10 ** decimals());
    }
}

// In TokenFactory.sol
contract TokenFactory is Ownable {
+   function deployToken(string memory symbol, bytes memory contractBytecode, address initialRecipient) public onlyOwner returns (address addr) {
-   function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner returns (address addr) {
+       bytes memory bytecodeWithInit = abi.encodePacked(contractBytecode, abi.encode(initialRecipient));
        assembly {
```

```
-           addr := create(0, add(contractBytecode, 0x20), mload(
    contractBytecode))
+           addr := create(0, add(bytecodeWithInit, 0x20), mload(
    bytecodeWithInit))
        }
        s_tokenToAddress[symbol] = addr;
        emit TokenDeployed(symbol, addr);
    }
}
```

[H-6] Use of Era-VM Incompatible Opcode in TokenFactory (Non-Compatible Assembly + Deployment Failure on ZKSync)

Description:

The `deployToken` function in `TokenFactory.sol` uses the `create` opcode in assembly to deploy new token contracts. This opcode is not compatible with ZKSync Era's Era-VM, which has different opcodes and deployment mechanics, causing the function to fail when executed on ZKSync.

Impact:

The `TokenFactory` contract cannot deploy new token contracts on ZKSync Era, as the `create` opcode is unsupported. This prevents the contract from functioning as intended on ZKSync, limiting the bridge's interoperability and deployment scope, and potentially causing deployment failures or unexpected behavior.

Proof of Concept:

In `TokenFactory.sol`:

```
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);
}
```

No test is needed, as the issue is a compilation/deployment failure on ZKSync Era due to the incompatible `create` opcode, as noted in the ZKSync documentation: Differences with Ethereum.

Recommended Mitigation:

Use ZKSync's system contract `CREATE` or a higher-level Solidity approach to ensure compatibility:

```
contract TokenFactory is Ownable {
+    import { IContractDeployer } from "@matterlabs/zksync-contracts/
    interfaces/IContractDeployer.sol";
+    address constant ZKSYNC_DEPLOYER = 0
    x0000000000000000000000000000000000008006;

     function deployToken(string memory symbol, bytes memory
         contractBytecode) public onlyOwner returns (address addr) {
-        assembly {
-            addr := create(0, add(contractBytecode, 0x20), mload(
    contractBytecode))
-        }
+        if (block.chainid == 324 || block.chainid == 300) { // ZKSync
    Mainnet or Testnet
+            addr = IContractDeployer(ZKSYNC_DEPLOYER).create("",
    contractBytecode);
+        } else {
+            assembly {
+                addr := create(0, add(contractBytecode, 0x20), mload(
    contractBytecode))
+            }
+        }
         s_tokenToAddress[symbol] = addr;
         emit TokenDeployed(symbol, addr);
     }
}
```

MEDIUM

[M-1] Malicious Data in sendToL1 Can Approve Vault Tokens to Attacker (Unrestricted Call Data + Vault Drainage)

Description:

The `sendToL1` function in `L1BossBridge.sol` allows a signed message to include arbitrary call data, which can target the `approveTo` function in `L1Vault.sol`. A malicious user can craft a signed message to approve themselves to transfer tokens from the vault, enabling them to drain the vault's token balance.

Impact:

A malicious user with a valid signature can approve themselves to withdraw all tokens from the vault, leading to complete drainage of the vault's funds. This compromises the bridge's security, as the vault is meant to securely hold tokens for L1-L2 bridging, resulting in significant financial loss.

Proof of Concept:

The issue is demonstrated in `L1BossBridgeTest.sol`:

```
function testCanCallApproveToAndStealTokensWithSendToL1Function()
    public {
     address maliciousUser = makeAddr("maliciousUser");
     uint256 initialVaultBalance = 1000 ether;
     deal(address(token), address(vault), initialVaultBalance);

     bytes memory message = abi.encode(
         address(vault),
         0,
         abi.encodeCall(L1Vault.approveTo, (maliciousUser, type(uint256)
             .max))
     );

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

     tokenBridge.sendToL1(v, r, s, message);

     vm.expectEmit(address(tokenBridge));
     emit Deposit(address(vault), maliciousUser, initialVaultBalance);
     tokenBridge.depositTokensToL2(address(vault), maliciousUser,
        initialVaultBalance);
}
```

Recommended Mitigation:

Restrict the `sendToL1` function to only allow calls to the token's `transferFrom` function:

```
contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
     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();
```

```
        }
        (address target, uint256 value, bytes memory data) = abi.decode
            (message, (address, uint256, bytes));
+       if (target != address(token) || bytes4(data) != IERC20.
    transferFrom.selector) {
+           revert L1BossBridge__Unauthorized();
+       }
        (bool success,) = target.call{ value: value }(data);
        if (!success) {
            revert L1BossBridge__CallFailed();
        }
    }
}
```

[M-2] Malicious Data in sendToL1 Can Cause Excessive Gas Consumption (Unrestricted Call Data + Gas Griefing)

Description:

The `sendToL1` function in `L1BossBridge.sol` allows a signed message to include arbitrary call data, which can be crafted to execute computationally expensive operations. A malicious user can submit data that consumes excessive gas, forcing the bridge (and signers indirectly) to incur high gas costs when processing the transaction.

Impact:

A malicious user can exploit the unrestricted call data to grief the bridge by submitting transactions that consume excessive gas, increasing operational costs for the bridge operator and potentially disrupting service by making withdrawals prohibitively expensive or causing transactions to fail due to gas limits.

Proof of Concept:

In `L1BossBridge.sol`:

```
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();
    }
```

```
    (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();
    }
}
```

Appending this test in `L1BossBridgeTest.sol` demonstrates the issue:

```
function testHighGasConsumptionInSendToL1() public {
    address maliciousUser = makeAddr("maliciousUser");
    address gasHeavyContract = address(new GasHeavyContract());
    bytes memory expensiveData = abi.encodeWithSignature("consumeGas()"
        );

    bytes memory message = abi.encode(gasHeavyContract, 0,
        expensiveData);
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(operator.key,
        MessageHashUtils.toEthSignedMessageHash(keccak256(message)));

    vm.prank(maliciousUser);
    uint256 gasUsed = gasleft();
    tokenBridge.sendToL1(v, r, s, message);
    assert(gasleft() < gasUsed - 1_000_000); // Significant gas
        consumption
}

// Helper contract to simulate gas-heavy operation
contract GasHeavyContract {
    function consumeGas() external {
        for (uint256 i = 0; i < 100000; i++) {
            keccak256(abi.encode(i));
        }
    }
}
```

Recommended Mitigation:

Limit the gas used in the `call` and restrict the target to the token contract:

```
contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
    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();
        }
```

```
        (address target, uint256 value, bytes memory data) = abi.decode(message, (address, uint256, bytes));
+       if (target != address(token) || bytes4(data) != IERC20.transferFrom.selector) {
+           revert L1BossBridge__Unauthorized();
+       }
-       (bool success,) = target.call{ value: value }(data);
+       (bool success,) = target.call{ value: value, gas: 100_000 }(data);
        if (!success) {
            revert L1BossBridge__CallFailed();
        }
    }
}
```

[L-1] Multiple Tokens allows Same Symbol Overwrite Previous Mapping (Lack of Symbol Uniqueness + Loss of Token Reference)

Description:

The `deployToken` function in `TokenFactory.sol` allows deploying multiple token contracts with the same symbol, overwriting the previous token address in the `s_tokenToAddress` mapping. This results in the loss of references to earlier deployed tokens with the same symbol, making them inaccessible via `getTokenAddressFromSymbol`.

Impact:

Overwriting token addresses in the `s_tokenToAddress` mapping causes earlier deployed tokens to become unreachable through the factory's lookup mechanism. This can lead to confusion, loss of access to previously deployed tokens, and potential operational issues in systems relying on the mapping for token identification.

Proof of Concept:

In `TokenFactory.sol`:

```
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);
}
```

Appending this test in `L1BossBridgeTest.sol` demonstrates the issue:

```
function testMultipleTokensWithSameSymbol() public {
    vm.startPrank(deployer);
    TokenFactory factory = new TokenFactory();
    bytes memory bytecode = type(L1Token).creationCode;

    address token1 = factory.deployToken("BBT", bytecode);
    address token2 = factory.deployToken("BBT", bytecode);

    assertEq(factory.getTokenAddressFromSymbol("BBT"), token2);
    assertNotEq(token1, token2);
    // token1 is no longer accessible via getTokenAddressFromSymbol
    vm.stopPrank();
}
```

Recommended Mitigation:

Prevent overwriting by checking if the symbol is already used in the mapping:

```
contract TokenFactory is Ownable {
+   error TokenFactory__SymbolAlreadyUsed();

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

[L-2] Lack of Event Emission in sendToL1 and withdrawTokensToL1 (Missing Event Logging + Reduced Transparency)

Description:

The `sendToL1` and `withdrawTokensToL1` functions in `L1BossBridge.sol` do not emit events upon successful execution. This lack of event emission reduces transparency, as off-chain services and users cannot easily track or verify withdrawals from L2 to L1.

Impact:

Without events, off-chain monitoring systems and users cannot reliably track withdrawal transactions, leading to reduced auditability and transparency. This may complicate debugging, monitoring, or integration with external systems, potentially affecting trust in the bridge's operations.

Proof of Concept:

In `L1BossBridge.sol`:

```
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(message)), 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();
     }
}
```

No test is needed, as the issue is the absence of event emission, which can be verified by inspecting the code.

Recommended Mitigation:

Add an event emission in `sendToL1` to log successful withdrawals:

```
contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
+   event Withdrawal(address indexed to, uint256 amount, address
    indexed target);

    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();
        }
        (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();
        }
+       if (target == address(token)) {
+           (, address to, uint256 amount) = abi.decode(data[4:], (
    address, address, uint256));
+           emit Withdrawal(to, amount, target);
+       }
    }
}
```

[L-3] Signer Disablement Causes Transaction Failure (Signer Status Change + Transaction Disruption)

Description:

The `setSigner` function in `L1BossBridge.sol` allows the owner to disable a signer. If a signer is disabled while a withdrawal transaction is in progress (e.g., after a signature is issued but before `sendToL1` is called), the transaction will fail due to the `signers[signer]` check, as the signer's status is no longer valid.

Impact:

Disabling a signer mid-flight can cause legitimate withdrawal transactions to revert, disrupting user operations and potentially locking funds temporarily until a new signature is obtained. This reduces the reliability of the bridge and may frustrate users expecting their transactions to complete.

Proof of Concept:

In `L1BossBridge.sol`:

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

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();
    }
    ...
}
```

Appending this test in `L1BossBridgeTest.sol` demonstrates the issue:

```
function testSignerDisablementCausesFailure() public {
    vm.startPrank(user);
    uint256 depositAmount = 10e18;
    token.approve(address(tokenBridge), depositAmount);
    tokenBridge.depositTokensToL2(user, userInL2, depositAmount);

    bytes memory message = _getTokenWithdrawalMessage(user,
        depositAmount);
    (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.
        key);

    vm.startPrank(deployer);
    tokenBridge.setSigner(operator.addr, false);
    vm.stopPrank();

    vm.prank(user);
    vm.expectRevert(L1BossBridge.L1BossBridge__Unauthorized.selector);
    tokenBridge.withdrawTokensToL1(user, depositAmount, v, r, s);
}
```

Recommended Mitigation:

Add a grace period or versioning for signer changes to allow pending transactions to complete:

```
contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
+   struct SignerStatus {
+       bool enabled;
+       uint256 updatedAt;
+   }
```

```
+   mapping(address => SignerStatus) public signers;
+   uint256 public constant SIGNER_GRACE_PERIOD = 1 days;

    function setSigner(address account, bool enabled) external onlyOwner {
-       signers[account] = enabled;
+       signers[account] = SignerStatus(enabled, block.timestamp);
    }

    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]) {
+       SignerStatus memory status = signers[signer];
+       if (!status.enabled && block.timestamp > status.updatedAt + SIGNER_GRACE_PERIOD) {
            revert L1BossBridge__Unauthorized();
        }
        ...
    }
}
```