Protocol Audit Report

Version 1.0

pureGavin

September 15, 2025

Protocol Audit Report

pureGavin

2025-09-15

Prepared by: [pureGavin] Lead Auditors: - pureGavin

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] Overly trusting the parameter address in `L1BossBridge::depositTokensToL2` may result in fund loss.
    * [H-2] `L1BossBridge::depositTokensToL2` Since `vault` was already `approve` during the initial phase, the contract can continuously transfer tokens to itself.
    * [H-3] The signature for `L1BossBridge::withdrawTokensToL1` lacks an expiration period, which may lead to signature reuse vulnerabilities.

    * [H-4] Incorrect `zkSync` call method for `TokenFactory::deployToken`
  - Informational
    * [I-1] For internal functions that are not used externally, it is recommended to change their visibility from **`public`** to `external`.
    * [I-2] Variables that never change during use can be declared as `constant` or `immutable`.

Protocol Summary

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

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

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

The owner of the bridge can pause operations in emergency situations. Because deposits are permissionless, there's an strict limit of tokens that can be deposited. Withdrawals must be approved by a bridge operator. We plan on launching L1BossBridge on both Ethereum Mainnet and ZKSync.

Disclaimer

The pureGavin team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M

		Impact		
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

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

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
High	4
Medium	
Low	
Informational	2
total	6

Findings

High

[H-1] Overly trusting the parameter address in `L1BossBridge::depositTokensToL2` may result in fund loss.

Description: The `depositTokensToL2` function does not validate the `l2Recipient` parameter before executing the transfer. This flaw allows attackers to deposit arbitrary amounts into the `vault`, provided the `from` party consents.

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

Impact: It is important to note that `from` consent is not a reliable form of verification, as consent is an integral part of the contract. If `from` does not consent, the contract's functionality cannot be utilized, meaning this vulnerability will inevitably be triggered.

Proof of Concept:

Poc code

Place the following code in `L1TokenBridge.t.sol`:

```
1      function testDepositTokensToL2() public {
2          vm.prank(user);
```

```
3           token.approve(address(tokenBridge), type(uint256).max);
4
5           uint256 depositAmount = token.balanceOf(user);
6           address attacker = makeAddr("attacker");
7           vm.startPrank(attacker);
8           vm.expectEmit(address(tokenBridge));
9           emit Deposit(user, attacker, depositAmount);
10          tokenBridge.depositTokensToL2(user, attacker, depositAmount);
11
12          console2.log("balance of user: ", token.balanceOf(user));
13          console2.log("balance of vault: ", token.balanceOf(address(
                vault)));
14          vm.stopPrank();
15      }
```

```
1  Logs:
2    balance of user:  0
3    balance of vault:  100000000000000000000
```

Recommended Mitigation:

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

[H-2] `L1BossBridge::depositTokensToL2` Since `vault` was already `approve` during the initial phase, the contract can continuously transfer tokens to itself.

Description: This vulnerability is similar to the previous one, with the difference being that this vulnerability involves transferring one's own `token` to oneself.

```
1       constructor(IERC20 _token) Ownable(msg.sender) {
2           token = _token;
3           vault = new L1Vault(token);
4   @>      vault.approveTo(address(this), type(uint256).max);
5       }
6   .
7   .
8   .
9       function depositTokensToL2(address from, address l2Recipient,
            uint256 amount) external whenNotPaused {
```

```
10          if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
11              revert L1BossBridge__DepositLimitReached();
12          }
13  @>      token.safeTransferFrom(from, address(vault), amount);
14
15          emit Deposit(from, l2Recipient, amount);
16      }
```

Impact: Since the contract was `approved` right from the start, triggering this vulnerability is extremely easy. The `BossBridge` contract generates its own `BossBridgeToken` after a successful deposit. Unlimited self-transfers mean `BossBridgeToken` can be generated without restriction.

Proof of Concept:

Poc code

Place the following code in `L1TokenBridge.t.sol`:

```
1       function testTransferVaultToVault() public {
2           address attacker = makeAddr("attacker");
3
4           uint256 vaultBalance = 500e18;
5           deal(address(token), address(vault), vaultBalance);
6
7           vm.expectEmit(address(tokenBridge));
8           emit Deposit(address(vault), address(attacker), vaultBalance);
9           tokenBridge.depositTokensToL2(address(vault), attacker,
                vaultBalance);
10
11          vm.expectEmit(address(tokenBridge));
12          emit Deposit(address(vault), address(attacker), vaultBalance);
13          tokenBridge.depositTokensToL2(address(vault), attacker,
                vaultBalance);
14      }
```

Recommended Mitigation:

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

[H-3] The signature for `L1BossBridge::withdrawTokensToL1` lacks an expiration period, which may lead to signature reuse vulnerabilities.

Description: The `withdrawTokensToL1` function uses an `ECDSA` signature when calling `sendToL1`, but does not validate the signature's validity.

```
1  @>  function withdrawTokensToL1(address to, uint256 amount, uint8 v,
       bytes32 r, bytes32 s) external {
2          sendToL1(
3              v,
4              r,
5              s,
6              abi.encode(
7                  address(token),
8                  0, // value
9                  abi.encodeCall(IERC20.transferFrom, (address(vault), to
                       , amount))
10             )
11         );
12     }
```

Impact: Since signature information is visible to anyone on-chain, functions without signature validity verification are extremely dangerous. In the `withdrawTokensToL1` function, this would directly result in the loss of all funds.

Proof of Concept:

Poc code

Place the following code in `L1TokenBridge.t.sol`:

```
1      function testSigReplay() public {
2          address attacker = makeAddr("attacker");
3          uint256 vaultInitBalance = 1000e18;
4          uint256 attackerInitBalance = 100e18;
5          deal(address(token), address(vault), vaultInitBalance);
6          deal(address(token), address(attacker), attackerInitBalance);
7          console2.log("balance of vault: ", token.balanceOf(address(
               vault)));
8          console2.log("balance of attacker: ", token.balanceOf(address(
               attacker)));
9
10         vm.startPrank(attacker);
11         token.approve(address(tokenBridge), type(uint256).max);
12         tokenBridge.depositTokensToL2(attacker, attacker,
               attackerInitBalance);
13
14         bytes memory message = abi.encode(address(token), 0, abi.
               encodeCall(IERC20.transferFrom, (address(vault), attacker,
               attackerInitBalance)));
```

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

        while(token.balanceOf(address(vault)) > 0) {
            tokenBridge.withdrawTokensToL1(attacker,
                attackerInitBalance, v, r, s);
        }
        console2.log("balance of vault: ", token.balanceOf(address(
            vault)));
        console2.log("balance of attacker: ", token.balanceOf(address(
            attacker)));
    }
```

```
// before
            Logs:
                balance of vault:  1000e18
                balance of attacker:  100e18

// after
            Logs:
                balance of vault:  0
                balance of attacker:  1100e18
```

Recommended Mitigation:

```
    mapping(address account => bool isSigner) public signers;
+   mapping(address => uint256) public nonces;

-   function withdrawTokensToL1(address to, uint256 amount, uint8 v,
    bytes32 r, bytes32 s) external {
+   function withdrawTokensToL1(address to, uint256 amount, uint8 v,
    bytes32 r, bytes32 s, uint256 deadline) external {
+       require (block.timestamp <= deadline, "Signature expired");
+       uint256 nonce = nonces[msg.sender];
+
+       bytes32 digest = keccak256(abi.encodePacked(to, amount, nonce,
    deadline));
+
+       address signer = ecrecover(digest, v, r, s);
+       require(signer == msg.sender, "Invalid signature");
+
+       nonces[msg.sender]++;

        sendToL1(
            v,
            r,
            s,
            abi.encode(
                address(token),
                0, // value
```

```
23                 abi.encodeCall(IERC20.transferFrom, (address(vault), to
                       , amount))
24             )
25         );
26     }
```

[H-4] Incorrect zkSync call method for `TokenFactory::deployToken`

Description: As described in the zkSyncofficial documentation, the following implementation approach will cause the protocol to fail to function properly.

```
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      }
```

Informational

[I-1] For internal functions that are not used externally, it is recommended to change their visibility from `public` to `external`.

Recommended Mitigation:

```
1  -     function deployToken(string memory symbol, bytes memory
       contractBytecode) public onlyOwner returns (address addr) {
2  +     function deployToken(string memory symbol, bytes memory
       contractBytecode) external onlyOwner returns (address addr) {
3  -     function getTokenAddressFromSymbol(string memory symbol) public
       view returns (address addr) {
4  +     function getTokenAddressFromSymbol(string memory symbol) external
       view returns (address addr) {
```

[I-2] Variables that never change during use can be declared as `constant` or `immutable`.

Recommended Mitigation:

```
1  -     uint256 public DEPOSIT_LIMIT = 100_000 ether;
2  -     IERC20 public token;
3  +     uint256 public constant DEPOSIT_LIMIT = 100_000 ether;
```

```
4 +     IERC20 public immutable token;
```