ThunderLoan Audit Report

Version 1.0

Zkillua.io

Boss-Bridge Audit Report

Zkillua.io

Feb 20th, 2024

Prepared by: [Zkillua]

Table of Contents

Table of Contents

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.

Disclaimer

The Zkillua 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

Scope

- Code Base:https://github.com/Cyfrin/7-boss-bridge-audit
- Commit Hash: 07af21653ab3e8a8362bf5f63eb058047f562375
- In Scope:

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

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.

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	3
Info	0
Gas Optimizations	0
Total	9

Findings

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

Description: 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).

Proof of Concept: Add the following test in `L1BossBridge.t.sol`

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

```

```
6       uint256 depositAmount = token.balanceOf(user);
7       vm.startPrank(attacker);
8       vm.expectEmit(address(tokenBridge));
9       emit Deposit(user, attacker, depositAmount);
10      tokenBridge.depositTokensToL2(user, attacker, depositAmount);
11
12      assertEq(token.balanceOf(user), 0);
13      assertEq(token.balanceOf(address(vault)), depositAmount);
14      vm.stopPrank();
15  }
```

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

```
1
2   -- function depositTokensToL2(address from, address l2Recipient,
       uint256 amount) external whenNotPaused {
3   ++ function depositTokensToL2(address l2Recipient, uint256 amount)
       external whenNotPaused {
4       if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
5           revert L1BossBridge__DepositLimitReached();
6       }
7   --    token.transferFrom(from, address(vault), amount);
8   ++    token.transferFrom(msg.sender, address(vault), amount);
9
10      // Our off-chain service picks up this event and mints the
           corresponding tokens on L2
11  --    emit Deposit(from, l2Recipient, amount);
12  ++    emit Deposit(msg.sender, l2Recipient, amount);
13
14  }
```

[H-2] 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:

```
1   function testCanReplayWithdrawals() public {
2       address attacker=makeAddr("attacker");
```

```
3
4       // Assume the vault already holds some tokens
5       uint256 vaultInitialBalance = 1000e18;
6       uint256 attackerInitialBalance = 100e18;
7       deal(address(token), address(vault), vaultInitialBalance);
8       deal(address(token), address(attacker), attackerInitialBalance);
9
10      // An attacker deposits tokens to L2
11      vm.startPrank(attacker);
12      token.approve(address(tokenBridge), type(uint256).max);
13      tokenBridge.depositTokensToL2(attacker, attacker,
            attackerInitialBalance);
14
15      // Operator signs withdrawal.
16      (uint8 v, bytes32 r, bytes32 s) =
17          _signMessage(_getTokenWithdrawalMessage(attacker,
                attackerInitialBalance), operator.key);
18
19      // The attacker can reuse the signature and drain the vault.
20      while (token.balanceOf(address(vault)) > 0) {
21          tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance
                , v, r, s);
22      }
23      assertEq(token.balanceOf(address(attacker)), attackerInitialBalance
            + vaultInitialBalance);
24      assertEq(token.balanceOf(address(vault)), 0);
25  }
```

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

[H-3] 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:

```
1  function testCanTransferFromVaultToVault() public {
2      vm.startPrank(attacker);
3
```

```
4       // assume the vault already holds some tokens
5       uint256 vaultBalance = 500 ether;
6       deal(address(token), address(vault), vaultBalance);
7
8       // Can trigger the `Deposit` event self-transferring tokens in the
            vault
9       vm.expectEmit(address(tokenBridge));
10      emit Deposit(address(vault), address(vault), vaultBalance);
11      tokenBridge.depositTokensToL2(address(vault), address(vault),
            vaultBalance);
12
13      // Any number of times
14      vm.expectEmit(address(tokenBridge));
15      emit Deposit(address(vault), address(vault), vaultBalance);
16      tokenBridge.depositTokensToL2(address(vault), address(vault),
            vaultBalance);
17
18      vm.stopPrank();
19  }
```

[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:

```
1  function test_attackerCanWithdrawAnyAmount() public{
2
3          address attacker=makeAddr("attacker");
4          uint256 vaultInitialBalance = 1000e18;
5          uint256 attackerInitialBalance = 1000e18;
```

```
6           deal(address(token), address(vault), vaultInitialBalance);
7           deal(address(token), address(attacker), attackerInitialBalance)
                ;
8
9
10          vm.startPrank(attacker);
11
12          (uint8 v, bytes32 r, bytes32 s) =
13          _signMessage(_getTokenWithdrawalMessage(attacker,
                attackerInitialBalance), operator.key);
14          console2.log("initial vault balance ",token.balanceOf(address(
                vault)));
15      // The attacker can reuse the signature and drain the vault.
16      while (token.balanceOf(address(vault)) > 0) {
17          tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance
                , v, r, s);
18      }
19          // tokenBridge.withdrawTokensToL1(attacker,vault_balance,v,r,s)
                ;
20          console2.log("vault balance ",token.balanceOf(address(vault)));
21
22      }
```

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

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`

[L-3] Unsupported opcode PUSH0