ackee blockchain

LayerZero

solidity-examples

by Ackee Blockchain

July 27, 2022

Contents

1. Document Revisions

1.0	Final report	July 27, 2022

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specialized in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run a free certification course Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, Rockaway Blockchain Fund.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Slither is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzzy testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests.

2.3. Review team

Member's Name	Position
Lukáš Böhm	Lead Auditor
Jan Šmolík	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.4. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Solidity-examples repository is used as a template or example for creating Omnichain Fungible (OFT) and Non-Fungible Tokens (ONFT) based on LayerZero's Interoperability protocol.

LayerZero engaged Ackee Blockchain to perform a security review of the Solidity-examples contracts with a total time donation of 7 engineering days in a period between July 15 and June 26, 2022 and the lead auditor was Lukáš Böhm.

The scope of the security review has been specified to contracts changed in two pull requests merged in the `audit` branch.

Commit: `c7525a549a8db3fb54a89620409bf29c89f23899`

We began our review by using static analysis tools. Then we took a deep dive into the logic of the contracts. During the review, we paid particular attention to:

- detecting possible reentrancies in the code,
- ensuring the proper handling of the tokens during the cross-chain messages,
- ensuring access controls are not too relaxed or too strict,
- looking for common issues such as data validation.

Our review resulted in 5 findings, ranging from Warning to Medium severity.

The architecture of the project is well designed and allows easy integration for 3th parties. The tests are written in JavaScript, and they successfully pass. Code quality is excellent and well documented, essential for the example contracts. LayerZero provides high-quality documentation in the

white paper and on the gitbook website.

Ackee Blockchain recommends LayerZero:

- use static analysis tools like `Slither`,
- ensure that the privileged owner role is well maintained,
- address all the reported issues.

4. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

4.1. Contracts

Contracts we find essential for better understanding are described in the following section.

The audit scope consists of two almost identical directories, one of which contains upgradeable versions of the contracts from the other directory.

LzApp.sol

LzApp is a base contract for Layer Zero protocol applications. These applications send and receive messages through `lzEndpoint`, which is set in the constructor. All the following tokens discussed in detail are themselves LzApp.

NonblockingLzApp.sol

NonblockingLzApp inherits from `LzApp` and implements three additional functions for receive handling and message retry.

OFT20.sol

OFT20 or `OmnichainFungibleToken` is an `ERC20` and `ERC165` token. It has built-in functionality to transfer itself to different blockchains supported by LayerZero.

OFT20Core.sol

OFT20Core is an abstract contract that contains the core functionality of

OFT20, which is a transfer to different blockchains supported by LayerZero.

BasedOFT20.sol

BasedOFT20 is an extension of OFT20, which can be minted in any quantity regardless of the chain. The whole BasedOFT20 token supply is minted on the base chain. Tokens transferred out of the base chain are locked in the contract and minted on destination. Tokens returning are burned on the destination and unlocked on the base chain.

GlobalCappedOFT20.sol

GlobalCappedOFT20 is an extension of OFT20 that adds a global cap to the supply of tokens across all chains.

PausableOFT20.sol

PausableOFT20 is an extension of OFT20 where the owner of the token contract has the right to pause transfers.

ProxyOFT20.sol

ProxyOFT20 is a proxy contract for OFT20. The exact token is set in the constructor and cannot be changed.

ONFT721Core.sol

ONFT721Core is an abstract contract that contains the core functionality of ONFT721. ONFT721 is an ERC721 and ERC165 token with built-in functionality to transfer itself to different Layer Zero supported blockchains.

ONFT721.sol

The contract contains two essential functions: _debitFrom and _creditTo, that burn or mint tokens for a specific user. In the upgradeable version

ONFT721Upgradeable.sol, there is no burning, but tokens are sent to this address instead. The token is sent to the end user if the contract holds the token with a given id. If the token with id does not exist, then it is minted.

ONFT1155Core.sol

ONFT1155Core is an abstract contract that contains the core functionality of ONFT1155. ONFT1155 is an ERC1155 and ERC165 token with built-in functionality to transfer itself to different Layer Zero supported blockchains. Unlike ONFT721 this contract can also send batches.

contracts-upgradable directory

This directory contains the Upgradeable versions of most contracts discussed above. Upgradeable contracts include two additional functions for the upgradeable pattern.

4.2. Actors

This part describes the system's actors, roles, and permissions.

Owner of the token contract

The owner of the token contract can:

- call setUseCustomAdapterParams() function to use custom adapter parameters in _send() function;
- in case of PausableOFT20, he can call pauseSendTokens() to pause transfers;
- call functions which change user application config in LzApp and its upgradable version LzApp.

User

A user can use the protocol to send cross-chain messages, check the total

supply of a specific token, or get information about trusted remotes.

4.3. Trust model

Users have to trust the owner of upgradable contracts that he will only perform an upgrade to a trusted contract.

Additionally, they have to trust the relayer and the oracle not to falsify the messages.

5. Vulnerabilities risk methodology

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood* which measures the probability of exploitability during runtime.

5.1. Finding classification

The full definitions are as follows:

Severity

Severity	Impact	Likelihood
Informational	Informational	N/A
Warning	Warning	N/A
Low	Low	Low
Medium	Low	Medium
Medium	Low	High
Medium	Medium	Medium
High	Medium	High
Medium	High	Low

Severity	Impact	Likelihood
High	High	Medium
Critical	High	High

Table 1. Severity of findings

Impact

High

Code that activates the issue will lead to undefined or catastrophic consequences for the system.

Medium

Code that activates the issue will result in consequences of serious substance.

Low

Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.

Warning

The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as "Warning" or higher, based on our best estimate of whether it is currently exploitable.

Info

The issue is on the border-line between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or

configuration (see above) was to change.

Likelihood

High

The issue is exploitable by virtually anyone under virtually any circumstance.

Medium

Exploiting the issue currently requires non-trivial preconditions.

Low

Exploiting the issue requires strict preconditions.

6. Findings

This section contains the list of discovered findings. Unless overriden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*, and
- a *Recommendation*

Many times, there might be multiple ways to solve or alleviate the issue, with varying requirements in terms of the necessary changes to the codebase. In that case, we will try to enumerate them all, making clear which solve the underlying issue better (albeit possibly only with architectural changes) than others.

Summary of Findings

	Severity	Impact	Likelihood
M1: Renounce ownership	Medium	High	Low
M2: Dangerous transfer ownership	Medium	High	Low
W1: Lack of events in state changing functions	Warning	Warning	N/A
W2: Usage of `solc` optimizer	Warning	Warning	N/A
W3: Floating pragma	Warning	Warning	N/A

Table 2. Table of Findings

M1: Renounce ownership

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	{LZApp.sol, LZAppUpgradeable.sol}	Type:	Access Control

Description

`LZApp.sol` (`LZAppUpgradeable.sol`) inherits functionality to renounce the ownership from `Ownable` (`OwnableUpgradeable`) contract. Function `renounceOwnership` sets the owner address to zero, and ownership is forever invalidated. However, the owner's role is necessary to maintain the contract's state and configuration.

Exploit scenario

Accidentally, the current owner calls `renounceOwnership` function, which forever invalidates the role. After that, nobody can get the ownership role and change the application configuration, set custom adapter parameters, or set a trusted remote.

Recommendation

Override the `renounceOwnership` method to disable this unwanted feature.

Go back to Findings Summary

M2: Dangerous transfer ownership

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	{LZApp.sol, LZAppUpgradeable.sol}	Type:	Access Control

Description

`LZApp.sol` (`LZAppUpgradeable.sol`) inherits functionality to transfer the ownership from `Ownable` (`OwnableUpgradeable`) contract.

```
function _transferOwnership(address newOwner) internal virtual {
        address oldOwner = _owner;
        _owner = newOwner;
        emit OwnershipTransferred(oldOwner, newOwner);
    }
```

However, the transfer function has **not** have a robust verification mechanism for the new proposed owner. If a wrong owner address is accidentally passed to it, the error cannot be recovered. Thus passing a wrong address can lead to irrecoverable mistakes.

Exploit scenario

The current owner Alice wants to transfer the ownership to Bob. Alice calls the `transferOwnership` function but supplies the wrong address by mistake. As a result, the ownership will be passed to a wrong and possibly dead address.

Recommendation

One of the common and safer approaches to ownership transfer is to use a two-step transfer process.

Current owner Alice wants to transfer the ownership to Bob. The two-step process would have the following steps: Alice proposes a new owner, namely Bob. This proposal is saved to a variable candidate. The candidate Bob calls the function for accepting the ownership, which verifies that the caller is the new proposed candidate. If the verification passes, the function sets the caller as the new owner. If Alice proposes the wrong candidate, she can change it. However, it can happen, with a very low probability that the wrong candidate is malicious (most often, it would be a dead address).

An authentication mechanism can also be employed to prevent the malicious candidate from accepting the ownership.

Go back to Findings Summary

W1: Lack of events in state changing functions

Impact:	Warning	Likelihood:	N/A
Target:	{LzApp.sol, LzAppUpgradeable.sol, *Core.sol}	Type:	Logging

Description

Besides the essential functions for message handling, LzApp contracts also log state changes while adding a new trusted remote in function `setTrustedRemote`. However, the state-changing function `setMinDstGasLookup` does not emit the change, although it updates the minimal gas limit.

```
function setMinDstGasLookup(uint16 _dstChainId, uint _type, uint
_dstGasAmount) external onlyOwner {
    require(_dstGasAmount > 0, "LzApp: invalid _dstGasAmount");
    minDstGasLookup[_dstChainId][_type] = _dstGasAmount;
}
```

All the `*Core.sol` contracts implement function `setUseCustomAdapterParams` for setting custom adapter parameter. Variable `useCustomAdapterParams` is global and changes contracts behavior, but this change is not logged.

```
function setUseCustomAdapterParams(bool _useCustomAdapterParams) external
onlyOwner {
        useCustomAdapterParams = _useCustomAdapterParams;
    }
```

Recommendation

Log any values that on-chain and off-chain observers might be interested in, particularly when the contract's state is changed. This ensures the maximum transparency of the protocol to its users, developers, and other stakeholders.

It may also help with a potential incident analysis.

Go back to Findings Summary

W2: Usage of `solc` optimizer

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Compiler configuration

Description

The project uses `solc` optimizer. Enabling `solc` optimizer may lead to unexpected bugs.

The Solidity compiler was audited in November 2018, and the audit concluded that the optimizer may not be safe.

One recent bug was discovered in June 2022. The bug caused some assembly memory operations to be removed, even though the values were used later in the execution.

Vulnerability scenario

A few months after deployment, a vulnerability is discovered in the optimizer. As a result, it is possible to attack the protocol.

Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt-out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

Go back to Findings Summary

W3: Floating pragma

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Compiler configuration

Description

The project uses solidity floating pragma: `^0.8.0`. Floating pragma allows using all the higher versions of the solidity compiler.

Vulnerability scenario

A mistake in deployment can cause a version mismatch and thus an unexpected bug. Floating pragma also allows higher vulnerable versions (0.8.14) to be used in the project, which can also cause unintended behavior.

Recommendation

Stick to one version, lock the pragma in all contracts, or set the range more precisely to avoid using unsafe versions. More information can be found at: swcregistry

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, LayerZero: solidity-examples, July 27, 2022.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

A non-`view` and non-`pure` function.

Appendix C: Theory of Upgradeability

This appendix lays out the most common pitfalls of upgradeability in Solidity and Ethereum generally, as well as our general recommendations. This is primarily for education purposes; for recommendations about the reviewed project, please refer to the **Recommendation** section of each finding.

Upgradeability is a difficult topic, and clsoe to impossible to get right. Most commonly, the following are the greatest issues in any upgradeability mechanism:

1. Interacting with the logic contract. If the syntactic contract or any of its superclasses contain a delegatecall or selfdestruct instruction, it could be possible to destroy the logic contract. See the Parity 2 vulnerability.
2. Front-running the initialization function (syntactically in the logic contract) on the proxy. This is $usually$ a non-issue and easy to solve, just check that the init function has not been called in the smart contract, and finally require the init function have succeeeded in the deployment script (e.g. JavaScript or Python).
3. Front-running other functions (syntactically in the logic contract) on the proxy. This can be a problem if the deployment of the proxy and the initialization of the logic contract are not atomic. In that case, it may be possible for an attacker to front-run the initialization, and call some function. If there are no access controls in this case, the call would succeed without the deployment script failing, and the attacker might have a backdoor in to the contract without anyone realizing it.

There tend to be two possible solutions for (1) and (3):

C.1. Manual review

For (1), this involves checking that it, or any of its ancestors, don't contain the delegatecall or selfdestruct instructions.

For (3), this involves checking that if any mutating function is called before the init function has been called, the call would not change any state in the system, e.g. by reverting.

C.2. Programmatic approach

The best way to accomplish both (1) and (3) (while preserving (2)) is to:

1. Ensure that no function on the deployed logic contract can be called until the initialization function is called.
2. Make sure that once the logic contract is constructed, its initialization function cannot be called.
3. Ensure that the initialization function can be called on the oroxy.
4. All functions can be called on the proxy once it has been initialized.

As an example, here is a way to accomplish that on a primitive upgradeable contract:

1. Ensure the initialization function can only be called once, e.g. by using OpenZeppelin's `initializer` modifier (see Listing 2).
2. Also add the `initializer` modifier to the constructor of the logic contract (see Listing 3).
    - The constructor is called on the deployed logic contract, but not on the proxy.
3. Add a `initialized` state variable (Listing 1) that gets set to `true` on initialization (see Listing 2). Note that we have to define a new variable,

since OpenZeppelin's `_initialized` is marked as `private`.

4. Add a require to every mutating function in the logic contract that it has been initialized (see Listing 4).

Listing 1. For syntactic logic contract

```
bool public initialized;
```

Listing 2. For syntactic logic contract

```
function initialize() public initializer {
    initialized = true;
}
```

Listing 3. For syntactic logic contract

```
constructor() initializer {}
```

Listing 4. For every mutating function in the syntactic logic contract

```
modifier onlyInitialized() {
    require(initialized);
    _;
}
```

In summary, the process would be to:

1. Add a requirement to every mutating function that the contract has been initialized.
2. Add a requirement to the initialization function that it cannot be called on the logic contract.

Together, these will accomplish both (1) and (3) of the upgradeability requirements.

Appendix D: Theory of Re-entrancy

This appendix discusses the theory of re-entrancy vulnerabilities. We'll first create a taxonomy (categorization) of re-entrancy attacks, discuss some proposed mitigations, and lastly give our general recommendation. This is primarily for education purposes; for recommendations about the reviewed project, please refer to the **Recommendation** section of each finding.

D.1. Introduction

A re-entrancy vulnerability might occur when an external call separates two code blocks, and somewhere on the network there is code that is contingent (dependent) on both blocks executing without interruption.

Notably, there are two ways to categorize re-entrancy attacks:

1. what caused the call?
    - a direct call (or ether transfer) to the attacker,
    - a call to another contract that calls (or transfers ether) to the attacker,
        - a special case of this are callback tokens, e.g. ERC223, ERC721, ERC777, ERC1155.
2. what was called in the re-entrancy?
    - it could be the same function (e.g. The DAO),
    - it could be a different function,
    - most importantly, it could be a different contract.

D.2. Proposed solutions

Checks-effects-interactions

A common recommendation to prevent re-entrancies is to use the "checks-effects-interactions" pattern. Roughly, this means:

1. First performing "checks", i.e. data validation.
2. Then performing "effects", data manipulation and storage writing.
3. Lastly performing "interactions", calling other contracts (including sending ether).

The intuition here is that an external call that occurs as the last statement in a function is not vulnerable to a re-entrancy since there is no code after it.

In practice it is rarely possible to fully follow this pattern. If a function needs to make multiple calls (e.g. `transferFrom` user and `transfer` to user in an Amm), there could be a re-entrancy in the first call.

Furthermore, it is very common for a function to write state based on an external call (for example, to an oracle) so state updates can't always precede external calls.

Re-entrancy locks

Another proposed solution is a contract-specific re-entrancy lock. (CSRL). While it can fix some re-entrancy attacks, complex DeFi systems are often composed of multiple modules. Re-entering another module may be a security risk.

For example, consider an Amm that works as follows:

- each *pool* holds stock of a single token, possibly callback tokens
- each *pair* allows to trade between two *pools*, i.e., between two tokens using the constant-product formula (`k == x*y == L^2`).

Even if all mutating function in all *pool*s and *pair*s are protected with a re-entrancy lock, the protocol can still be attacked:

- a user swaps *token1* for *token0*
- whichever token transfer happens first (either from user to pool or from pool to user), they can re-enter into *a different* pair. At this point, the balances of the protocol are in a dirty, re-entered state (in particular, the constant product formula does not apply) and the attacker can use that to gain an unfair amount of tokens.

D.3. General recommendations

Tokens

In the example, the only reason it was possible to execute a re-entrancy was that the token transfer allowed a re-entrancy into the system. Unless there is a strong reason in favor, avoid callback tokens, such as ERC223, ERC721, ERC777, ERC1155. These tokens make every token transfer an opportunity for code injection and makes security analysis much more difficult.

Notably, avoid these tokens for things like:

- native protocol tokens (LP token template, governance token, Ico token, etc.),
- whitelisting these tokens (e.g. as collateral tokens, reserve tokens, reward tokens etc.),
- allowing users to add these tokens in any way that could impact the system,
- include in your documentation that behavior is undefined if users add these tokens in they're forks / trustless submodules (e.g. an Amm pair).

System-wide re-entrancy lock (SWRL)

Another reason why the above attack was feasible was that the re-entrancy locks only applied to a single contract. Indeed, this should not be a surprise. Each module having its own re-entrancy lock is equivalent to partitioning the system's functions and applying a re-entrancy lock only on each partition. Just like that isn't safe in a single-contract scenario, it isn't safe in a multi-contract scenario.

For complex systems with multiple modules, it is usually favorable to have a system-wide re-entrancy lock (SWRL). The idea is that at any point in time, we are executing just one untrusted call. However, we also need contracts *in the system* to be able to call each other without triggering the lock, so the lock needs to allow trusted addresses to have multiple calls into the system.

Putting these two things together yields the following semantics:

1. There should be a registry of all system smart contracts.
2. When a contract receives a call from another smart contract, it is allowed to go through.
3. When a contract receives a call from an untrusted address, then require that the lock has not been acquired (and acquire it).

This is the only way to protect against cross-module re-entrancy attacks (CMRAs).

ackee

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://discord.gg/z4KDUbuPxq