Layer Zero

Solidity-examples

by Ackee Blockchain

10.10.2022

Contents

1. Document Revisions

1.0	Final report	July 27, 2022
2.0	Final re-audit report	October 10, 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, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School School of Solana, 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. Finding classification

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.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Medium	-
	Low	Medium	Medium	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

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 a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline 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.

2.4. Review team

Member’s Name	Position
Lukáš Böhm	Lead Auditor
Miroslav Škrabal	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. 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.

3.1. Revision 1

LayerZero engaged Ackee Blockchain to perform a security review of the Solidity-examples contracts with a total time donation of 10 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 was specified to contracts changed in two pull requests merged in the `audit` branch.

Commit: `c7525a5`

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 3rd 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.

3.2. Revision 2

LayerZero provided an updated codebase with new contracts on September 26, 2022. The review was done on the given commit: `4fc0dc7`.

Security review was between September 26 and October 7, 2022. The lead auditor was Lukáš Böhm.

We used static analysis tools during the review, including the new Woke tool with built-in automated vulnerability detectors. Detectors of Woke found issues such as event ordering . Our review resulted in 11 findings, ranging from `informational` to `medium` severity. From previous review only M1: Lack of emits was fixed, the rest remain relevant.

See Revision 2.0 for the review of the updated codebase and additional information we consider essential for the current scope. Summary of Findings provides a status of each issue.

Implemented Woke tests for BytesLib library can bee seen in Appendix B.

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a `Description`,
- an `Exploit scenario`,
- a `Recommendation` and if applicable
- a `Solution`.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
M1: Renounce ownership	Medium	1.0	Reported
M2: Dangerous transfer ownership	Medium	1.0	Reported
W1: Lack of events in state changing functions	Warning	1.0	Fixed
W2: Usage of `solc` optimizer	Warning	1.0	Reported
W3: Floating pragma	Warning	1.0	Reported
M3: Initialize functions in upgradeability	Medium	2.0	Reported
M4: Address validation in constructor	Medium	2.0	Reported

	Severity	Reported	Status
M5: Data validation in constructor	Medium	2.0	Reported
M6: Accepting messages from untrusted remotes	Medium	2.0	Reported
W4: Lack of existence check	Warning	2.0	Reported
W5: Order of event emission and reentrancy	Warning	2.0	Reported
W6: Lack of cross-chain token ID validation	Warning	2.0	Reported
W7: Mint without price require	Warning	2.0	Reported
I1: Gas optimization in mint function	Info	2.0	Reported
I2: Unused SafeERC20	Info	2.0	Reported
I3: Bad coding practices	Info	2.0	Reported

Table 2. Table of Findings

5. Report revision 1.0

5.1. 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.

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.

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.

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.

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.

Revision 2.0

One of the contracts that inherits from [LZApp](#) is [pausable](#). Thus renouncing ownership can block the app forever when the owner is renounced while the [contract](#) is paused.

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.

Solution (Revision 2.0)

Emits have been added.

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.

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: `^0.8.2`.

Vulnerability scenario

A mistake in deployment can cause a version mismatch and thus an unexpected bug.

Recommendation

Stick to one version and lock the pragma in all contracts. More information can be found in: SWC Registry.

Go back to Findings Summary

6. Report revision 2.0

6.1. System Overview

The updated scope contains a `composable` folder with a set of composable contracts, where `ComposableBasedOFT.sol` is a copy of BasedOFT.sol (`OFT20` contracts have been renamed to `OFT` in version `2.0`) and `ComposableOFT.sol` is a copy of OFT.sol. Contract `ComposableOFTCore.sol` extends OFTCore.sol

UniversalONFT721.sol

Universal example of `ONFT721` with implemented `mint` function with the ability to mint on different chains thanks to the ID ranges.

ExcessivelySafeCall.sol

The library with upgraded low-level call functions. The primary and key difference is the ability to limit the number of bytes to copy to the caller contract memory. A user can use this functionality by setting `_maxCopy` argument.

BytesLib.sol

The external library implements functions for byte manipulation. It can compare the storage of bytes, concat, slice, or convert bytes into a specific data type.

NativeOFT.sol

The contract with implemented core functionalities like `send`, `withdraw`, `receive` and `creditTo`. It is and extension of OFT that allows for wrapping the native currency and thus making it possible to transfer it cross-chain

M3: Initialize functions in upgradeability

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	contracts-upgradeable/*.sol	Type:	Upgradeability

Listing 1. Excerpt from OFTCoreUpgradeable.__OFTCoreUpgradeable_init

```
    function __OFTCoreUpgradeable_init(address _endpoint) internal
onlyInitializing {
        __OFTCoreUpgradeable_init_unchained(_endpoint);
    }
```

Listing 2. Excerpt from OFTCoreUpgradeable.__OFTCoreUpgradeable_init_unchained

```
    function __OFTCoreUpgradeable_init_unchained(address _endpoint)
internal onlyInitializing {
        __NonblockingLzAppUpgradeable_init_unchained(_endpoint);
    }
```

Description

The LayerZero upgradeable contracts use the `_init` and `_init_unchained` functions as known from OpenZeppelin upgradeability. Those functions are meant to initialize the contract state and avoid double initialization. The `_init` function should perform the logic that would typically be done in the constructor header, and the `_init_unchained` should perform the logic that would be done in the constructor body.

However, this pattern is violated, which can lead to double initialization. This issue generally applies to the upgradeable contracts, but for simplicity will be illustrated on the `OFTCoreUpgradeable` contract's initialization.

The `OFTCoreUpgradeable` contains both the `_init` and `_init_unchained` functions. The `_init` function only performs a call to the `_init_unchained` (see Listing 1). The `_init_unchained` performs a call to the `_init_unchained` function of the `LzApp` (see Listing 2). The `_init_unchained` function of the `OFTCoreUpgradeable` is thus chained (it chains the call to the `_init_unchained` function of the `LzApp`).

Vulnerability scenario

A developer of a user application follows the initialization pattern. To avoid double initialization, he calls all the `_init_unchained` functions of the contracts he derives his app from (while using a properly linearized call sequence). This approach is valid because no `_init_unchained` should perform chaining. However, the `OFTCoreUpgradeable` violates this pattern, and thus the developer will call the `_init_unchained` function of the `LzApp` twice (once from his contract, once from the `OFTCoreUpgradeable`), which will lead to double initialization.

Recommendation

Follow the pattern that the `_init` function acts as a constructor header and `_init_unchained` as a constructor body. The `_init_unchained` function should not perform chaining. The `_init` function should contain linearized calls to the `_init_unchained` functions of all the contracts it derives from. Such an approach assures that double initialization will not happen and that all variables will get initialized. For inspiration contracts from OpenZeppelin can be used.

Go back to Findings Summary

M4: Address validation in constructor

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	LzApp.sol	Type:	Data validation

Description

The LzApp contract does not perform any data validation on the `_endpoint` argument.

Exploit scenario

An incorrect value of `_endpoint` is passed to the LzApp constructor. Instead of reverting, the call succeeds. If such a mistake is not discovered quickly and the contracts are not redeployed, the protocol can behave in an undefined way.

Recommendation

Add more stringent data validation for `_endpoint`. At the very least, this would include a zero-address check. Ideally, it is recommended to define a getter in the `_endpoint` such as `contractId()` that would return a hash of an identifier unique to the (project, contract). An example would be `keccak256("Layer Zero Endpoint")`. Upon constructing the `LZApp` the identifier would be retrieved from the passed-in `_endpoint` address and compared to the expected value.

Go back to Findings Summary

M5: Data validation in constructor

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	UniversalONFT721.sol	Type:	Data validation

Description

The UniversalONFT721 contract does not perform validation whether `_endMintId` is greater than `_startMintId`

Exploit scenario

Value `_startMintId` is set with higher number than `_endMintId` in UniversalONFT721 constructor. If such a case happened, the contract would become unusable.

Recommendation

In the case of UniversalONFT721, a simple condition should be included.

```
require(_startMintId < _endMintId, "Invalid mint interval")
```

Go back to Findings Summary

M6: Accepting messages from untrusted remotes

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	LzApp.sol, NonblockingLzApp.sol	Type:	Data validation

Description

The LzApp contract uses `trustedRemoteLookup` mapping to validate whether a given remote can be trusted or not. The values to the mapping are set by the contract owner in the function `setTrustedRemote`. If special conditions are met, a call from an untrusted remote can be executed. The `retryMessage` function enables this as it performs no check on whether the given remote is trusted or not. Additionally, it is enabled by the fact that a given remote can be set as untrusted after it is set as trusted.

Vulnerability scenario

1. Owner sets a new trusted remote `A` using `setTrustedRemote` function.
2. `A` gets compromised by a malicious actor.
3. A message from `A` is delivered. The execution fails inside the `_blockingLzReceive,` and the message is thus stored to `failedMessages` mapping.
4. Because `A` was compromised, the owner sets `A` as untrusted. He does so by setting the value for key `A` to the default value.
5. The failed message gets reexecuted by calling `retryMessage`, and this time the call succeeds. As a result, a message from an *untrusted* remote gets executed.

Recommendation

Inside the `retryMessage` function, implement an additional check whether or not a given remote is trusted. This check is needed because a remote can become untrusted while a message is stored in the `failedMessages` buffer.

[Go back to Findings Summary](#)

W4: Lack of existence check

Impact:	Warning	Likelihood:	N/A
Target:	**/*	Type:	Data validation

Listing 3. Excerpt from ComposableOFTCore._safeCallOnOFTReceived

```
    function _safeCallOnOFTReceived(uint16 _srcChainId, bytes memory
_srcAddress, uint64 _nonce, bytes memory _caller, bytes memory _from,
address _to, uint _amount, bytes memory _payload, uint _gasForCall)
internal virtual {
        (bool success, bytes memory reason) = _to.excessivelySafeCall
(_gasForCall, 150, abi.encodeWithSelector(IOFTReceiver.onOFTReceived
.selector, _srcChainId, _srcAddress, _nonce, _caller, _from, _amount,
_payload));
        if (!success) {
```

Description

The function `_safeCallOnOFTReceived` performs a low level call on the `_to` address (see Listing 3). The function performs no existence check on the `_to` address. Currently, the function is only called from a place that performs this check. However, such a check should be performed in the function itself.

Exploit Scenario

Alice, a user application developer, derives her contract from the `ComposableOFTCore`. Assuming that the function `safeCallOnOFTReceived` is entirely safe (as implied by its name), Alice uses the function without validating the `_to` address. A call to the function is made with an address that does not correspond to a contract, and the call falsy succeeds.

Recommendation

Perform the check directly in `_safeCallOnOFTReceive` function. This would make the function truly safe and provide the benefit of not having to perform the check-in functions that call it.

Go back to Findings Summary

W5: Order of event emission and reentrancy

Impact:	Warning	Likelihood:	N/A
Target:	ComposableOFTCore.sol	Type:	Logging, Reentrancy

Listing 4. Excerpt from ComposableOFTCore.retryOFTReceived

```
        delete failedOFTReceivedMessages[_srcChainId][_srcAddress
    ][_nonce];
        IOFTReceiver(_to).onOFTReceived(_srcChainId, _srcAddress,
    _nonce, _srcCaller, _from, _amount, _payload);
        emit RetryOFTReceivedSuccess(hash);
```

Description

The `retryOFTReceived` function may log events in the wrong order in case of reentrancy.

The event `RetryOFTReceivedSuccess` is emitted only after the call to `onOFTReceived()` function (see Listing 4). The call to `onOFTReceived()` can result in reentering the contract via the same function. In such a case, the inner calls would emit the events first (the first in, last out principle), which might break assumptions about the event emission order.

Recommendation

Follow the Checks-Effects-Interactions Pattern even for logging.

Go back to Findings Summary

W6: Lack of cross-chain token ID validation

Impact:	Warning	Likelihood:	N/A
Target:	UniversalONFT721.sol	Type:	Data validation

Description

The UniversalONFT721 uses id ranges to allow for safe minting. Each chain has an id range that does not intersect with other chains. This mechanism is used against minting NFTs with identical ids on different chains. The requirement for unique ranges is not enforced on the smart-contract level, as this would be rather complicated.

Exploit scenario

If this constraint was violated, it could lead to critical mistakes. One such mistake is DoS of the mint function. Suppose that chains `A` and `B` have intersecting id ranges. An nft from colliding range with index i is transferred from `A` to `B`. Once the id to be minted on `B` is i, minting gets DoSed. This is because the _mint function in ERC721 requires:

```
require(!_exists(tokenId), "ERC721: token already minted");
```

Recommendation

It must be ensured that the deployment script for those contracts is validated and audited appropriately to avoid the possibility of deploying contracts with intersecting id ranges.

Go back to Findings Summary

W7: Mint without price require

Impact:	Warning	Likelihood:	N/A
Target:	UniversalONFT721	Type:	Data validation

Description

In the contract, `mint` function is declared as s payable without any requirement on the price of NFT. Because of the `payable` keyword, users are expected to send Ether to the function to acquire NFTs. However, the function lacks any requirements on the price of NFT. On the one hand, this is logical because each project will use a different price for the NFT, but the requirement should be present for a good example code.

Recommendation

We recommend defining a constant representing the price:

```
unit256 public constant NFT_PRICE = 1;
```

and then add a requirement to the mint function:

```
require(msg.value >= NFT_PRICE, "insufficient ether");
```

Go back to Findings Summary

I1: Gas optimization in mint function

Impact:	Info	Likelihood:	N/A
Target:	UniversalONFT721.sol	Type:	Gas optimization

Description

In UniversalONFT721 in `mint` function post-inc is used:

```
nextMintId++;
```

Recommendation

Where the semantics of the program does not change, it is recommended to use pre-inc as it is more gas efficient. The `mint` function could be even simplified to:

```
function mint() external payable {
  require(nextMintId <= maxMintId, "UniversalONFT721: max mint limit
reached");
  _safeMint(msg.sender, nextMintId++);
}
```

Go back to Findings Summary

I2: Unused SafeERC20

Impact:	Info	Likelihood:	N/A
Target:	NativeOFT.sol	Type:	Unused library

Description

The NativeOFT contract uses the `SafeERC20` library for the `IERC20` interface. However, the library is never used.

Recommendation

Consider whether including the `SafeERC20` library is necessary and optionally remove it. In this specific case, no unverified tokens are handled. Thus there is no actual use case for the library.

Go back to Findings Summary

I3: Bad coding practices

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	coding practices

Description

Across the project, there are several code inconsistencies.

Not accurate NatSpec comment

The constructor of contract UniversalONFT721 uses NatSpec comments for input argument description. However, the comment states that `_endMintId` represents max number of mints but it actually represents the maximum NFT id.

Misleading comment

ExcessivelySafeCall contains `ExcessivelySafeCall` library. The library contains the function `excessivelySafeCall`. The function mainly limits the number of bytes returned to a caller from a low-level call. The limit is imposed by the caller and is represented as uint16. However, the function contains the following comment:

```
// limit our copy to 256 bytes
```

However, the copy is limited by the `_maxCopy`, which is `uint16`, and thus the comment is misleading.

Unresolved TODO

The contract NativeOFT contains TODO comment on line 11:

```
// todo: should inherit from OFT/OFTCore
```

We recommend to finish all the TODOs before the deployment to prevent unwanted behavior

Useless transfer

In the same contract in function `_debitMsgFrom`, there is a line of code minting the token to the `msg.sender` address and then immediately sending the token from `msg.sender` to `address(this)`. The same result can be achieved by direct minting to `address(this)`. Public payable function `_send` uses an underscore in its name, which indicates internal functions.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Layer Zero: Solidity-examples, 10.10.2022.

Appendix B: Woke fuzz tests

We implemented tests using Woke fuzzer for BytesLib functions `slice` and `toAddress` used in the project. Both tests passed `successfully`. Functions outputs are consistent with expected behavior even for the high number of test campaigns.

For information on how to use the tool or run the tests, we provide the official documentation.

B.1. slice

```python
import brownie
from random import randint
from woke.fuzzer import Campaign
from woke.fuzzer.decorators import (
    flow,
    ignore,
    invariant,
    max_times,
    precondition,
    weight,
)
from woke.fuzzer.random import (
    random_account,
    random_bool,
    random_bytes,
    random_int,
    random_string,
)

class TestingSequence:
    def __init__(self, SliceTest):
        self.slice_test = SliceTest.deploy({'from': random_account()})

    @flow
    def flow_slice(self):
        b = random_bytes(0, 100)
        start = random_int(0, len(b))
        #allow end to be greater than len(b) to test revert on out-of-
bounds
        length = random_int(0, len(b) - start + 10)
        if start + length > len(b):
            with brownie.reverts("slice_outOfBounds"):
                self.slice_test.slice(b, start, length)
        else:
            expected_res = b[start:start+length].hex()
            res = self.slice_test.slice(b, start, length)
            assert res == "0x" + expected_res


def test_string_bytes_utils(SliceTest):
    campaign = Campaign(lambda: TestingSequence(SliceTest))
    campaign.run(1, 10)
```

B.2. toAddress

```
import brownie
from brownie.convert import to_bytes
from random import choice
from woke.fuzzer import Campaign
from woke.fuzzer.decorators import (
    flow,
    ignore,
    invariant,
    max_times,
    precondition,
    weight,
)
from woke.fuzzer.random import (
    random_account,
    random_bool,
    random_bytes,
    random_int,
    random_string,
)

class TestingSequence:
    def __init__(self, ToAddressTest):
        self.adr_test = ToAddressTest.deploy({'from': random_account()})

    @flow
    def flow_to_address(self):
        a = random_account()
        a_bytes = bytes.fromhex(a.address[2:])
        pre = random_bytes(0, 20)
        post = random_bytes(0, 20)
        if choice([True, False]):
            #tst for correct address
            assert self.adr_test.toAddress(pre + a_bytes + post, len(pre))
== a.address
        else:
            if pre:
                #tst for wrong address - start converting at position 0, ie
from pre bytes
                assert self.adr_test.toAddress(pre + a_bytes + post, 0) !=
a.address
            elif post:
                #tst for out of bounds - set start to index after
sssssncoded address
                with brownie.reverts("toAddress_outOfBounds"):
                    self.adr_test.toAddress(pre + a_bytes + post, len(pre +
a_bytes) + len(post)/2)


def test_string_bytes_utils(ToAddressTest):
    campaign = Campaign(lambda: TestingSequence(ToAddressTest))
    campaign.run(30, 100)
```

Appendix C: 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.

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://discord.gg/z4KDUbuPxq