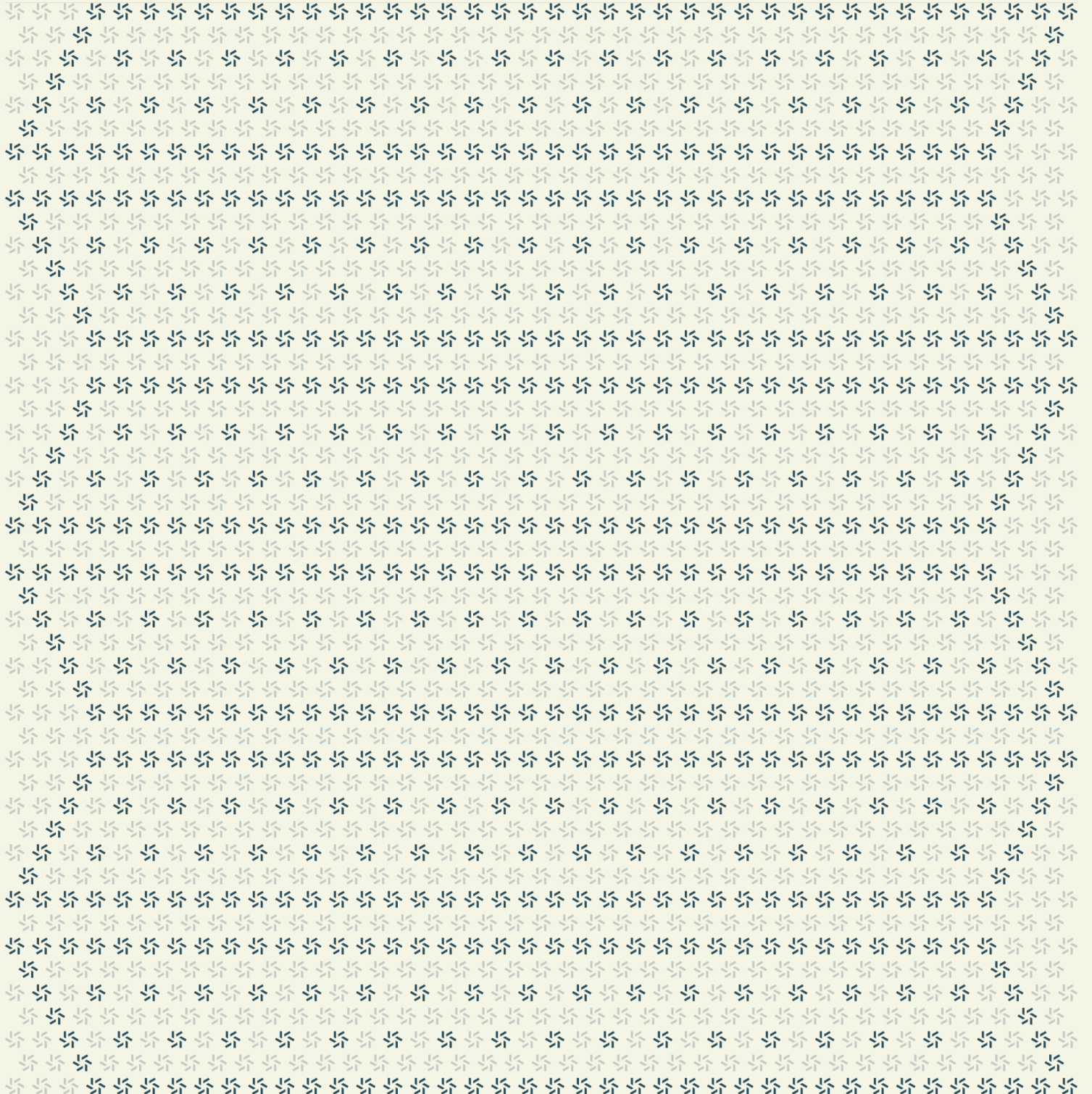


June 6, 2025

Example String-Passing Solana OApp

Smart Contract Security Assessment



Contents

About Zellic	4
<hr data-bbox="490 403 1563 407"/>	
1. Overview	4
1.1. Executive Summary	5
1.2. Goals of the Assessment	5
1.3. Non-goals and Limitations	5
1.4. Results	5
<hr data-bbox="490 785 1563 789"/>	
2. Introduction	6
2.1. About Example String-Passing Solana OApp	7
2.2. Methodology	7
2.3. Scope	9
2.4. Project Overview	9
2.5. Project Timeline	10
<hr data-bbox="490 1226 1563 1230"/>	
3. Detailed Findings	10
3.1. Initialization can be front-run	11
3.2. Ambiguous state initialization	12
<hr data-bbox="490 1486 1563 1491"/>	
4. Discussion	12
4.1. Various notes	13
4.2. Remediation review	13
<hr data-bbox="490 1747 1563 1751"/>	

5.	Threat Model	14
5.1.	Module: MyOApp.sol	15
5.2.	Program: Example OApp	17

6.	Assessment Results	25
6.1.	Disclaimer	26

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the [#1 CTF \(competitive hacking\) team](#) worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io and follow [@zellic_io](#) on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io.



1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for LayerZero Labs, Inc. from May 26th to May 28th, 2025. During this engagement, Zellic reviewed the example string-passing Solana OApp's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Was the cross-chain string-passing OApp example implemented correctly on both the EVM and Solana sides?
-

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

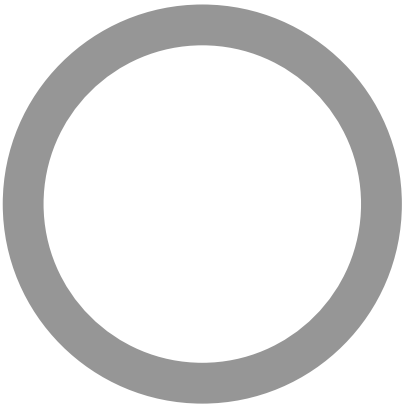
1.4. Results

During our assessment on the scoped example string-passing Solana OApp programs, we discovered two findings, both of which were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of LayerZero Labs, Inc. in the Discussion section ([4. 7](#)).

Breakdown of Finding Impacts

Impact Level	Count
<div>Critical</div>	0
<div>High</div>	0
<div>Medium</div>	0
<div>Low</div>	0
<div>Informational</div>	2



2. Introduction

2.1. About Example String-Passing Solana OApp

LayerZero Labs, Inc. contributed the following description of the example string-passing Solana OApp:

This is a simple cross-chain string-passing OApp example involving EVM and Solana.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the programs.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped programs itself. These observations — found in the Discussion (4. 7) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Example String-Passing Solana OApp Programs

Types	Solidity, Rust
Platforms	Solana, EVM-compatible
Target	devtools
Repository	https://github.com/LayerZero-Labs/devtools ↗
Version	8e518c61d12577c8bcb9c3cb5163dba0de087db2
Programs	contracts/**/*.sol programs/my_oapp/src/**/*.rs

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of three person-days. The assessment was conducted by two consultants over the course of three calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
↗ Engagement Manager
jacob@zellic.io ↗

Chad McDonald
↗ Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
↗ Engineer
kate@zellic.io ↗

Filippo Cremonese
↗ Engineer
fcremo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 26, 2025 Start of primary review period

May 28, 2025 End of primary review period

June 25, 2025 Start of secondary review period

June 25, 2025 End of secondary review period

3. Detailed Findings

3.1. Initialization can be front-run

Target	Example Solana OApp		
Category	Business Logic	Severity	Low
Likelihood	Low	Impact	Informational

Description

The `InitStore` instruction initializes a singleton PDA holding the OApp configuration and registers the OApp with LayerZero. The instruction also sets the admin for the OApp.

The instruction can only be invoked once, and it could be front-run if program deployment and OApp initialization are not performed in the same transaction.

Impact

The OApp initialization could be front-run, initializing the OApp with incorrect administrator and LayerZero endpoint addresses and requiring a redeployment of the program.

Recommendations

This low-likelihood threat is acceptable for most use cases; we recommend to consider documenting this as a potential issue for high-risk, high-value targets, such as DEXes or oracles.

Remediation

This finding has been acknowledged by LayerZero Labs, Inc., and was addressed in commit [833544ac](#) by adding a comment explaining the risk to the `InitStore` instruction.

3.2. Ambiguous state initialization

Target	Example Solana and EVM OApps		
Category	Business Logic	Severity	Low
Likelihood	Low	Impact	Informational

Description

Both the Solana and EVM example OApps save the received message in permanent storage.

When the OApps are initialized, the variable holding the received message is initialized with the default content "Nothing received yet.". This makes the state of newly initialized OApps indistinguishable from the state of an OApp that has received a message containing the same string.

Impact

The OApps cannot reliably distinguish between having received no message or having received a message with the content "Nothing received yet..".

As the OApps are just examples that do not implement any real functionality, this is not a vulnerability per se, but this design sets a poor example for developers who will use the example OApps as a starting point for their products.

Recommendations

The Solana OApp could use `Option<String>` to store the received message.

Solidity does not have a built-in option type; therefore, a separate boolean flag could be used.

Remediation

This issue has been acknowledged by LayerZero Labs, Inc..

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Various notes

The following notes do not constitute security issues and should be interpreted as observations with minor improvement suggestions for the codebase.

Anchor version

The version of the Anchor framework used by the codebase (v0.29.0) is not the latest as of the time of this writing (v0.31.0). We suggest using the updated version.

Comments referencing counter

Some comments or variable names refer to the singleton `Store` account as a "count" or "counter". This is likely because the codebase was derived from another example implementing a counter. This confusing naming could be improved.

Send options are not deduplicated

The message send options used by the `Send` instruction are formed by combining the enforced send options configured for the remote contract with the options provided with the instruction. The options are combined using `LZ combine_options`, which does not deduplicate repeated options. It is therefore possible for the outgoing message to contain repeated options. The LayerZero [documentation](#) explains the behavior in case options are repeated:

Passing identical `_options` in both `enforcedOptions` and `extraOptions` will cause the protocol to charge the caller twice on the source chain, because LayerZero interprets duplicate `_options` as two separate requests for gas.
















While allowing to specify repeated options is an intended feature, it could cause unexpected higher fees, and we suggest to document this behavior more prominently.

4.2. Remediation review

We performed a review of the remediations and additional changes made to the Solana OApp program (and corresponding Solidity contracts) made after this engagement on June 25th, 2025.

This follow-up review covered changes made from commit [8e518c61](#) to commit [7972f738](#), limited to the on-chain contracts/programs that were part of the original engagement scope. This

section describes the changes made in each individual commit.

- **Commit [fd66a009](#)** : Updates the README.
- **Commit [c2116580](#)** : Removes mentions of "count", addressing discussion point [4.1](#).
- **Commit [3d64aed2](#)** : Removes mentions of "count", addressing discussion point [4.1](#).
- **Commit [833544ac](#)** : Addresses finding [3.1](#), by adding a comment warning about the possibility to front-run program initialization.
- **Commit [0d4f9b4f](#)** : Updates the lockfile.
- **Commit [ed4eba41](#)** : Formats code in documentation.
- **Commit [593329a1](#)** : Renames parameters on the Solidity contract send function, for improved clarity.
- **Commit [488ad1cc](#)** : Switches Solidity contracts message decoder from using require to using custom errors.
- **Commit [c9a66a21](#)** : Updates .changeset.
- **Commit [cd491d2e](#)** : Updates dependencies for typescript client.
- **Commit [82a04bce](#)** : Removes a commented-out line introduced in the previous commit.
- **Commit [62031009](#)** : Bumps Typescript dependencies version.
- **Commit [aa21531c](#)** : Changes casing of my0app to myoapp for typescript clients.
- **Commit [6ce61d1b](#)** : Adds many comments to the Solana OApp, oriented towards helping OApp developers understand how the example OApp works and how it can be modified to build a custom OApp. Also contains one change to the Solidity contracts, renaming one `_payload` variable to `_message`.
- **Commit [7972f738](#)** : Reverts the casing change applied in [aa21531c](#), turning myoapp back to my0app.

The commits have not introduced new issues in the in-scope contracts/programs.

5. Threat Model

As time permitted, we analyzed each instruction in the program and created a written threat model for the most critical instructions. A threat model documents the high-level functionality of a given instruction, the inputs it receives, and the accounts it operates on as well as the main checks performed on them; it gives an overview of the attack surface of the programs and of the level of control an attacker has over the inputs of critical instructions.

For brevity, system accounts and well-known program accounts have not been included in the list of accounts received by an instruction; the instructions that receive these accounts make use of Anchor types, which automatically ensure that the public key of the account is correct.

Discriminant checks, ownership checks, and rent checks are not discussed for each individual account; unless otherwise stated, the program uses Anchor types, which perform the necessary checks automatically.

Not all instructions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that an instruction is safe.

5.1. Module: MyOApp.sol

Function: `send(uint32 _dstEid, string calldata _message, bytes calldata _options)`

This function allows any user to send any message to the application in the `_dstEid` chain.

Inputs

- `_dstEid`
 - **Control:** Full control.
 - **Constraints:** peers should contains the `_dstEid`. Only owner can set up.
 - **Impact:** The ID of the destination chain where the message will be sent.
- `_message`
 - **Control:** Full control.
 - **Constraints:** N/A.
 - **Impact:** The message to be sent.
- `_options`
 - **Control:** Full control.
 - **Constraints:** Should be `OPTION_TYPE_3`.
 - **Impact:** Determines the gas limit and amount of `msg.value` to include.

Branches and code coverage

Intended branches

- `_message` is sent successfully.
 - ☐ Test coverage
- The caller pays more fees than necessary and receives the excess back.
 - ☐ Test coverage

Negative behavior

- The provided fee is not enough.
 - ☐ Negative test
- The message is empty.
 - ☐ Negative test
- There is an invalid type of additional `_options`.
 - ☐ Negative test
- `_dstEid` is not supported.
 - ☐ Negative test

Function call analysis

- `_lzSend(_dstEid, _payload, options, MessagingFee(msg.value, 0), payable(msg.sender)) -> endpoint.send{ value: messageValue }(MessagingParams(_dstEid, _getPeerOrRevert(_dstEid), _message, _options, _fee.lzTokenFee > 0), _refundAddress);`
 - **What is controllable?** `_dstEid`.
 - **If the return value is controllable, how is it used and how can it go wrong?** `receipt` is just used to be returned from this function. Contains GUID, nonce, and fee.
 - **What happens if it reverts, reenters or does other unusual control flow?** If the provided fee is greater than necessary, the caller will receive excess back.
- `_lzSend(_dstEid, _payload, options, MessagingFee(msg.value, 0), payable(msg.sender)) -> _getPeerOrRevert(_dstEid)`
 - **What is controllable?** `_dstEid`.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returns the peer address where the message will be delivered.
 - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if `_dstEid` is not supported.

Function: `_lzReceive(Origin calldata _origin, bytes32 _guid, bytes calldata payload, address _executor, bytes calldata _extraData`

This function is part of the process of receiving a message from another chain. The main verification logic is performed in the `lzReceive` function, which triggers this internal function to decode the received payload and saved message. All parameters except `payload` are ignored.

Inputs

- `payload`
 - **Control:** Full control.
 - **Constraints:** The `payload.length` should be more than 32.
 - **Impact:** Contains the length of the message and message itself.

Branches and code coverage

Intended branches

- data is updated successfully by the new message.

☐ Test coverage

Negative behavior

- `payload.length` is less than 32.

☐ Negative test

5.2. Program: Example OApp

The Solana example OApp implements a basic OApp, which is able to perform cross-chain calls that pass a string between a Solana program and other allowlisted remote contracts using LayerZero.

The life cycle of the example OApp is simple. After the OApp program is deployed, it must be initialized and registered with LayerZero by using the `InitStore` instruction. The instruction also assigns the admin of the OApp.

The admin must then invoke `SetPeerConfig` to allowlist contracts on remote chains that are allowed to send messages to the OApp.

After this, messages can be sent by using `QuoteSend` to get an estimate of the required fees for a given message and destination and `Send` to actually send the message.

Messages received from other contracts are processed by the `LzReceive` instruction handler, which verifies that the incoming message is legitimate with the LayerZero contracts and then acts

on the incoming message.

Instruction: InitStore

This instruction initializes the example OApp. It initializes an account holding the configuration of the program and registers the OApp with the application with the LayerZero endpoint.

The instruction can only be invoked once for any given instance of the program.

Input parameters

```
pub struct InitStoreParams {
    pub admin: Pubkey,
    pub endpoint: Pubkey,
}
```

- **admin:** This is the address of the admin for this program.
- **endpoint:** This is the address of the LayerZero endpoint.

Accounts

The instruction receives the following accounts explicitly:

- **payer:** Account that pays account-creation fees.
 - Signer: Yes.
 - Mutable: Yes.
- **store:** Newly initialized Store account.
 - Init: Yes.
 - PDA: Yes, with fixed seed STORE_SEED.
- **lz_receive_types_accounts:** Newly initialized LzReceiveTypesAccounts account.
 - Init: Yes.
 - PDA: Yes, with seeds LZ_RECEIVE_TYPES_SEED and store.key().

It also forwards the remaining accounts to the CPI that registers the OApp with the LayerZero endpoint.

CPI

The instruction performs one CPI call to the LayerZero endpoint to invoke the RegisterOApp instruction.

This call is needed to initialize an OAppRegistry account, which stores the address of the [delegate](#) for the OApp.

The delegate can configure critical parameters of the OApp, including the security thresholds.

Instruction: SetPeerConfig

This instruction can be used by the OApp admin to initialize and edit PeerConfig configuration accounts representing an instance of an associated application on a remote chain. Multiple PeerConfig accounts can exist.

The instruction allows to edit one of the following configuration parameters at a time:

- `peer_address`: This is the address of the peer contract on the remote chain.
- `enforced_options`: This specifies the set of enforced [options](#) that are always added to outgoing messages destined to this peer.

Input parameters

```
pub struct SetPeerConfigParams {
    pub remote_eid: u32,
    pub config: PeerConfigParam,
}

pub enum PeerConfigParam {
    PeerAddress([u8; 32]),
    EnforcedOptions { send: Vec<u8>, send_and_call: Vec<u8> },
}
```

- `remote_eid`: This is the arbitrary ID identifying the remote contract.
- `config`: This is the parameter to be configured.

Accounts

- **admin**: Authorizes the operation and pays rent for account creation.
 - Signer: Yes.
 - Mutable: Yes.
 - Constraints: Must be the admin configured in the store.
- **peer**: Account representing the configuration for the remote contract.
 - Init: Yes, if needed.
 - PDA: Yes, with seeds `PEER_SEED`, `store.key()`, and `params.remote_eid`.

- **store:** Account storing the OApp global configuration.
 - PDA: Yes, with seed STORE_SEED.

Instruction: QuoteSend

This instruction returns a quote of the fees needed to send a cross-chain message.

Input parameters

```
pub struct QuoteSendParams {
  pub dst_eid: u32,
  pub receiver: [u8; 32],
  pub message: String,
  pub options: Vec<u8>,
  pub pay_in_lz_token: bool,
}
```

- **dst_eid:** This is the identifier of the remote endpoint.
- **receiver:** This is the address of the receiver contract.
- **message:** This is the arbitrary string to be sent to the remote contract.
- **options:** These are the send options; these are used in addition to the enforced options specified in the OApp configuration for the given receiver.
- **pay_in_lz_token:** If this is false, the fee is calculated in native currency; otherwise, it is calculated using the LayerZero token.

Accounts

The instruction receives the following accounts explicitly:

- **store:** Account storing the global OApp configuration.
 - PDA: Yes, with seed STORE_SEED.
- **peer:** Account representing an allowlisted remote contract.
 - PDA: Yes, with seeds PEER_SEED, store.key(), and params.dst_eid.
- **endpoint:** Account storing the settings for the LayerZero endpoint on Solana.
 - PDA: Yes, with seed ENDPOINT_SEED, derived from the endpoint program ID.

It also forwards the remaining accounts in the CPI made to get the messaging fee from the LayerZero endpoint.

CPI

The LayerZero endpoint is invoked to get a quote for the messaging fees.

The accounts for the CPI are forwarded verbatim from the Anchor context `remaining_accounts`, and the caller is responsible for providing the correct ones.

The `QuoteParams` struct providing arguments for the instruction is initialized as follows:

- `sender`: `ctx.accounts.store.key()`.
- `dst_eid`: `params.dst_eid`.
- `receiver`: `params.receiver`.
- `message`: `params.message` encoded by prepending the string length (32 bytes) before the UTF-8 representation of the string.
- `pay_in_lz_token`: `params.pay_in_lz_token`.
- `options`: `params.options` combined with the enforced options configured in the per-peer configuration.

Instruction: Send

This instruction sends a cross-chain message to one of the allowlisted remote contracts.

Input parameters

```
pub struct SendMessageParams {
    pub dst_eid: u32,
    pub message: String,
    pub options: Vec<u8>,
    pub native_fee: u64,
    pub lz_token_fee: u64,
}
```

- `dst_eid`: This is the ID of the remote endpoint contract.
- `message`: This is the message to be sent to the remote contract.
- `options`: These are additional send options, merged with the enforced options configured for the `dst_eid` in the remote peer configuration.
- `native_fee`: This is the amount of native currency to be used to pay for messaging fees.
- `lz_token_fee`: This is the amount of LZ token to be used to pay for messaging fees.

Accounts

- **peer**: Account representing the allowlisted remote contract to which the message is sent.
 - PDA: Yes, with seeds `PEER_SEED`, `store.key()`, and `params.dst_eid`.
- **store**: Account storing the OApp global configuration.
 - PDA: Yes, with seed `STORE_SEED`.
- **endpoint**: Account storing the settings for the LayerZero endpoint on Solana.
 - PDA: Yes, with seed `ENDPOINT_SEED`, derived from the endpoint program ID.

CPI

The LayerZero endpoint program is invoked to send the outgoing message.

The `SendParams` struct of arguments passed to the CPI is initialized as follows:

- `dst_eid`: `params.dst_eid`.
- `receiver`: `ctx.accounts.peer.peer_address`.
- `message`: `params.message` encoded by prepending the string length (32 bytes) before the UTF-8 representation of the string.
- `options`: `params.options` appended to the options enforced by the OApp global configuration.
- `native_fee`: `params.native_fee`.
- `lz_token_fee`: `params.lz_token_fee`.

Instruction: LzReceive

This instruction is invoked to process an incoming message. The call can be performed by anyone. The OApp invokes the LayerZero endpoint to verify that the incoming message is legitimate and is not being replayed.

If the endpoint CPI succeeds, the incoming message is decoded and stored in the singleton `Store` account, after which execution ends.

Input parameters

The instruction receives as argument an instance of the `LzReceiveParams` struct defined by the LayerZero SDK:

```
pub struct LzReceiveParams {  
    pub src_eid: u32,
```

```
pub sender: [u8; 32],
pub nonce: u64,
pub guid: [u8; 32],
pub message: Vec<u8>,
pub extra_data: Vec<u8>,
}
```

- `src_eid`: This is the identifier of the source endpoint.
- `sender`: This is the address of the remote message sender.
- `nonce`: This is the nonce used to prevent message replay and out-of-sequence delivery.
- `guid`: This is the globally unique (across all LayerZero OApps on all chains) identifier of the message.
- `message`: This is the incoming message payload.
- `extra_data`: This is unused.

Accounts

The instruction is intended to process the accounts returned by the `LzReceiveTypes` instruction.

The following accounts are directly required by the instruction:

- **store**: Singleton account storing the OApp configuration.
 - PDA: Yes, with seed `STORE_SEED`.
 - Mutable: Yes.
- **peer**: Account representing the configuration of one allowlisted remote contract.
 - PDA: Yes, with seeds `PEER_SEED`, `store.key()`, and `params.src_eid`.

The rest of the remaining accounts supplied to the instruction are passed to the LayerZero endpoint `Clear` CPI.

CPI

This instruction performs a CPI call to the LayerZero endpoint to invoke the `Clear` instruction, which verifies the incoming message and marks it as processed (advancing the nonce associated with the channel).

The `ClearParams` struct passed as argument to the CPI is initialized as follows:

- `receiver`: `ctx.accounts.store.key()`.
- `src_eid`: `params.src_eid`.
- `sender`: `params.sender`.
- `nonce`: `params.nonce`.
- `guid`: `params.guid`.

- `message: params.message.`

Instruction: LzReceiveTypes

This instruction returns the list of accounts needed by the `LzReceive` instruction to process a given message. It is used by message relayers to build a transaction that correctly invokes `LzReceive`. The list must include the accounts used by the OApp itself as well as accounts needed by the endpoint program `Clear` instruction.

The accounts needed by the example OApp are just the `Store` account holding the global configuration, which is a singleton with a fixed seed, and the account representing an allowlisted remote peer contract; the address of the account representing the remote peer is a PDA derived from the seeds `PEER_SEED`, `ctx.accounts.store.key()`, and `params.src_eid`.

The `Store` account is marked as writable, since the `LzReceive` instruction will write its data.

The list of accounts is extended by adding the accounts returned by the SDK function `get_accounts_for_clear`, which include (among others) the endpoint program, the OApp registry, and the nonce account used to prevent message-replay issues. The OApp itself does not need to be concerned with which specific accounts are necessary for LayerZero inner workings and does not need to validate the list returned by `get_accounts_for_clear`.

Input parameters

This instruction receives as argument an instance of the `LzReceiveParams` struct defined by the LayerZero SDK.

```
pub struct LzReceiveParams {
    pub src_eid: u32,
    pub sender: [u8; 32],
    pub nonce: u64,
    pub guid: [u8; 32],
    pub message: Vec<u8>,
    pub extra_data: Vec<u8>,
}
```

- `src_eid`: This is the ID of the sender endpoint.
- `sender`: This is the address of the sender of the message.
- `nonce`: This is used by `get_accounts_for_clear` to derive the PDA of the payload hash account needed by LayerZero internal logic.
- `guid`: This is unused.
- `message`: This is unused.
- `extra_data`: This is unused.

More sophisticated OApps may need to inspect the unused fields such as the message passed to

LzReceive to determine which accounts are needed.

Accounts

- **store:** Singleton account storing the OApp global configuration.
 - PDA: Yes, with seed STORE_SEED.

6. Assessment Results

During our assessment on the scoped example string-passing Solana OApp programs, we discovered two findings, both of which were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.