Layer Zero Security Review

Pashov Audit Group

Conducted by: Jakub Heba, defsec, 0xdeadbeef, ubermensch

May 15th 2024 - May 28th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **LayerZero-Labs/monorepo** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Layer Zero

LayerZero is an immutable messaging protocol designed to facilitate the creation of omnichain, interoperable applications. The scope included Layer Zero V2 for Solana.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 25c8b94370f63bc7b05d65f0471a2078779d3dc3

fixes review commit hash - a2e5ad2e582f98bb32d27ff774476a051b201cc5

Scope

The following smart contracts were in scope of the audit:

- `packages/layerzero-v2/solana/programs/blocked-messagelib/*`
- `packages/layerzero-v2/solana/programs/dvn/*`
- `packages/layerzero-v2/solana/programs/endpoint/*`
- `packages/layerzero-v2/solana/programs/executor/*`
- `packages/layerzero-v2/solana/programs/messagelib-interface/*`
- `packages/layerzero-v2/solana/programs/oft/*`
- `packages/layerzero-v2/solana/programs/pricefeed/*`
- `packages/layerzero-v2/solana/programs/uln/*`
- `packages/layerzero-v2/solana/programs/worker-interface/*`
- `packages/layerzero-v2/solana/libs/*`

7. Executive Summary

Over the course of the security review, Jakub Heba, defsec, 0xdeadbeef, ubermensch engaged with Layer Zero to review Layer Zero. In this period of time a total of **10** issues were uncovered.

Protocol Summary

Protocol Name	Layer Zero
Repository	https://github.com/LayerZero-Labs/monorepo
Date	May 15th 2024 - May 28th 2024
Protocol Type	Messaging protocol

Findings Count

Severity	**Amount**
High	1
Medium	2
Low	7
Total Findings	**10**

Summary of Findings

ID	Title	Severity	Status
[H-01]	A malicious actor can prevent sending messages through the executor and ULN	High	Resolved
[M-01]	Missing Accounts in lz_receive_types Return	Medium	Resolved
[M-02]	Insecure initialization functions	Medium	Acknowledged
[L-01]	Insecure Authority Transfer Functionality	Low	Acknowledged
[L-02]	Missing Events on Setter Functions	Low	Acknowledged
[L-03]	Missing Versioning in Solana OFT Program	Low	Resolved
[L-04]	Missing unique signer check in signers_setter function	Low	Resolved
[L-05]	Scalability Challenges in Cross-Chain Communication	Low	Acknowledged
[L-06]	Missing Duplicate Check in required_dvns and optional_dvns	Low	Resolved
[L-07]	Potential Impersonation of PDA Account in send Instruction of Endpoint Module	Low	Resolved

8. Findings

8.1. High Findings

[H-01] A malicious actor can prevent sending messages through the executor and ULN

Severity

Impact: High

Likelihood: Medium

Description

The `Executor` program has an `acl` that is checked against when calling `quote`:

```
impl Quote<'_> {
    pub fn apply
      (ctx: &Context<Quote>, params: &QuoteExecutorParams) -> Result<u64> {
        require!(!ctx.accounts.executor_config.paused, ExecutorError::Paused);
        let config = &ctx.accounts.executor_config;

        config.acl.assert_permission(&params.sender)?;
-----------------
```

The permission is revoked through the `SetDenylist` function. The function allows the caller to set addresses that will be included in the `deny_list` acl.

```
#[derive(Accounts)]
pub struct SetDenylist<'info> {
    pub owner: Signer<'info>,
    #[account(
        mut,
        seeds = [EXECUTOR_CONFIG_SEED],
        bump = config.bump
    )]
    pub config: Account<'info, ExecutorConfig>,
}

impl SetDenylist<'_> {
    pub fn apply(
      ctx:&mutContext<SetDenylist>,
      params:&SetDenylistParams
    ) -> Result<(
        for i in 0..params.denylist.len() {
            ctx.accounts.config.acl.set_denylist(&params.denylist[i])?;
        }
        Ok(())
    }
}
```

Notice that there is no access control and any user can call this function. A malicious actor can prevent access to any `sender` (OApp) that is used to call the executor quote function.

Currently, the `ULN` calls the `quote` function when paying the executor for a `send`.

```
fn pay_executor<'c: 'info, 'info>(
    uln: &Pubkey,
    payer: &AccountInfo<'info>,
    executor_config: &ExecutorConfig,
    dst_eid: u32,
    sender: &Pubkey,
    calldata_size: u64,
    options: Vec<LzOption>,

         accounts: &[AccountInfo<'info>], /* [executor_program, executor_config, price
                                           * price_feed_config] */
) -> Result<WorkerFee> {
    let fee =
        quote_executor(
          uln,
          executor_config,
          dst_eid,
          sender,
          calldata_size,
          options,
          accounts
        )?;
--------------
```

Therefore, a malicious actor can prevent execution through the ULN as long as they keep the deny list updated

Recommendations

Consider adding to the `config` account:

```
has_one = owner,
```

8.2. Medium Findings

[M-01] Missing Accounts in `lz_receive_types` Return

Severity

Impact: Low

Likelihood: High

Description

The `lz_receive_types` function is missing the `account 8 - associated token program` and `account 9 - system program` in its return value.

The `lz_receive_types` function is used to extract the accounts needed for calling `lz_receive` starting from `LzReceiveParams` as input.

The absence of these accounts will cause the `lz_receive` function to fail when it relies on the accounts returned by `lz_receive_types`.

```
File: programs/oft/src/instructions/lz_receive_types.rs#L70-L75

accounts.extend_from_slice(&[
    LzAccount { pubkey: receiver, is_signer: false, is_writable: false },
    LzAccount { pubkey: token_dest, is_signer: false, is_writable: true },
    LzAccount { pubkey: oft.token_mint, is_signer: false, is_writable: false },

        LzAccount { pubkey: oft.token_program, is_signer: false, is_writable: false }
]);
```

Recommendations

Include the missing accounts in the return value of the `lz_receive_types` function.

This will ensure that all necessary accounts are available for the successful execution of the `lz_receive` function.

[M-02] Insecure initialization functions

Severity

Impact: Medium

Likelihood: Medium

Description

Throughout the codebases, there are many initialization functions that do not have any access control. This means that an attacker can initialize them before the true caller does.

Some of these initialization functions have sensitive parameters that are set during the call such as admin keys. These include:

1. `ULN::init_uln`
2. `pricefeed::init_price_feed`
3. `oft::init_oft`
4. `oft::init_adapter_oft`
5. `executor::init_executor`
6. `endpoint::init_endpoint`
7. `dvn::init_dvn`

Here is an example of the ULN initialization that has no access control:

```
#[instruction(params: InitUlnParams)]
pub struct InitUln<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,
    #[account(
        init,
        payer = payer,
        space = 8 + UlnSettings::INIT_SPACE,
        seeds = [ULN_SEED],
        bump
    )]
    pub uln: Account<'info, UlnSettings>,
    pub system_program: Program<'info, System>,
}

impl InitUln<'_> {
    pub fn apply(ctx: &mut Context<InitUln>, params: &InitUlnParams) -> Result<
      ()> {
        ctx.accounts.uln.eid = params.eid;
        ctx.accounts.uln.endpoint = params.endpoint;
        ctx.accounts.uln.endpoint_program = params.endpoint_program;
        ctx.accounts.uln.admin = params.admin;
        ctx.accounts.uln.treasury = None;
        ctx.accounts.uln.treasury_fee_cap = params.treasury_fee_cap;
        ctx.accounts.uln.treasury_admin = params.treasury_admin;
        ctx.accounts.uln.bump = ctx.bumps.uln;
        Ok(())
    }
}
```

Since these init functions cannot be called directly at program deployment - an attacker can call them before the deployer and set arbitrary parameters (including admin keys). This might not be noticed by the deployer and the hacker could gain privileges to the program or it will be noticed and the program will be redeployed (maybe multiple times).

Recommendations

The deployer or other known address should be verified in the init function. Anchor docs suggest using the `upgrade_authority_address`: link

8.3. Low Findings

[L-01] Insecure Authority Transfer Functionality

The provided code snippet demonstrates an insecure implementation of an authority transfer functionality in a Solana program. The TransferAdmin struct and its associated apply function allow the current admin to directly transfer the admin role to a new address without any additional safeguards or validation.

The vulnerability lies in the fact that the current admin can unilaterally transfer the admin role to any arbitrary address without the consent or acceptance of the proposed new admin. This lack of a proper two-step transfer process exposes the program to several risks:

Accidental transfers: The current admin may inadvertently transfer the admin role to an incorrect or unintended address, leading to a loss of control over the program. **Malicious takeovers:** An attacker who compromises the current admin's private key can exploit this functionality to transfer the admin role to an address under their control, effectively taking over the program. **Lack of consent:** The proposed new admin has no means to accept or reject the transfer of authority, which violates the principle of explicit consent and may lead to unexpected responsibilities or liabilities.

```
impl TransferAdmin<'_> {
    pub fn apply(
      ctx:&mutContext<TransferAdmin>,
      params:&TransferAdminParams
    ) -> Result<(
        ctx.accounts.oft_config.admin = params.admin;
        Ok(())
    }
}
```

Note : The issue exists on the all authority transfers.

To mitigate this vulnerability and ensure a secure and consensual authority transfer process, it is recommended to implement a two-step transfer mechanism:

Nomination by the Current Admin:

1. Introduce a new instruction, such as nominate_new_admin, that allows the current admin to nominate a new admin address.
2. Update the OftConfig struct to include a pending_admin field to store the nominated address.
3. Modify the nominate_new_admin instruction to set the pending_admin field in the OftConfig account.

Acceptance by the Nominated Admin:

1. Introduce another instruction, such as accept_admin_role, that allows the nominated admin to explicitly accept the transfer of authority.
2. In the accept_admin_role instruction, verify that the signer is the pending_admin and update the admin field in the OftConfig account to the pending_admin address.
3. Clear the pending_admin field to indicate the completion of the transfer process.

[L-02] Missing Events on Setter Functions

In the given Solana program, there are functions that modify important program state variables, such as configuration settings or parameters. However, these state-modifying functions do not emit any events to signal the changes made to the program's state.

Code Locations :

- set_dst_config.rs#L21
- set_price_feed.rs#L18
- set_supported_option_types.rs#L21

To address the missing events issue, it is recommended to add event emission to the state-modifying functions.

[L-03] Missing Versioning in Solana OFT Program

The Solana OFT program, located at lib.rs, lacks versioning functionality compared to its EVM counterparts.

In the EVM smart contracts for LayerZero OFT, versioning is implemented to manage contract upgrades and ensure backward compatibility. Versioning allows contract developers to introduce new features, fix bugs, or make improvements while maintaining a clear distinction between different versions of the contract.

However, the Solana OFT program does not include any explicit versioning mechanism.

To address the missing versioning issue in the Solana OFT program, it is recommended to implement a versioning mechanism similar to the one used in the EVM smart contracts.

[L-04] Missing unique signer check in `signers_setter` function

It was observed that while initializing the DVM program, a check ensures only unique signers are added to the program, preventing operational issues caused by non-unique signers. However, a similar check is missing in the `signers_setter` function, which can be executed by the DVM authority to update the signers. The absence of this check can lead to non-unique signers being added, making the program non-operational.

During the initialization of the DVM program, the following code ensures that only unique signers are added:

```
impl InitDvn<'_> {
    pub fn apply(ctx: &mut Context<InitDvn>, params: &InitDvnParams) -> Result<
      ()> {
        assert_unique_signers(&params.signers)?;
        require!(
            params.quorum > 0 && params.quorum as usize <= params.signers.len(),
            DvnError::InvalidQuorum
        );
        // Initialization logic continues...
    }
}
```

This check prevents the inclusion of duplicate signers in the signers vector, which is crucial for the operational integrity of the program.

However, in the SetSigners function, which is used to update the signers, the check for unique signers is missing:

```
impl SetSigners<'_> {
    pub fn apply
      (ctx: &mut Context<SetSigners>, params: &SetSignersParams) -> Result<()> {
        //@audit-issue they should check if signers do not overlap - same as
        // they did in init
        ctx.accounts.config.multisig.signers = params.signers.clone();
        Ok(())
    }
}
```

Add the `assert_unique_signers` check in the `SetSigners` function.

[L-05] Scalability Challenges in Cross-Chain Communication

In LayerZero's Solana implementation, program events are emitted using Anchor's CPI-based log messages. This approach involves making a self-CPI call into the program with the serialized event as instruction data, allowing it to be persisted in the transaction and fetched later as needed. While this method provides a way to store and retrieve event data, it comes with certain limitations:

- Limited Message Size: The size of CPI calls is restricted to around 1.2kB, which limits the amount of data that can be included in the emitted log messages. This can impact the interoperability of the bridge with other chains that may require larger message payloads.
- Call Stack Depth Constraints: The use of self-CPI calls for log message emission contributes to the call stack depth, which is limited by Solana's runtime security measures. This can restrict the interoperability of the bridge with other programs and contracts that may require deeper call stack levels.

These limitations pose scalability challenges for LayerZero's Solana implementation, as they constrain the amount of log message data that can be stored and transmitted, and limit the bridge's ability to integrate with a wide range of contracts and chains.

To address the scalability and interoperability challenges, it is recommended to explore alternative approaches for storing log message contents. One potential

solution is to utilize dedicated storage accounts at deterministic addresses to store log message data.

[L-06] Missing Duplicate Check in `required_dvns` and `optional_dvns`

The `init_default_config` instruction lacks a check for duplicate entries in the `required_dvns` and `optional_dvns` vectors within the `UlnConfig`. This oversight could lead to redundant DVNs causing double voting on payload confirmations.

To address this, ensure to implement a validation step that checks for and prevents duplicate `Pubkey` entries in both `required_dvns` and `optional_dvns` before setting the default configuration. This can help maintain data integrity and avoid potential issues related to duplicate configurations.

[L-07] Potential Impersonation of PDA Account in send Instruction of Endpoint Module

Description

In the `simplemessage` lib, `SendWithLzToken` instruction accepts such accounts:

```
pub endpoint: Signer<'info>,
    pub message_lib: Account<'info, MessageLib>,
    pub message_lib_lz_token: InterfaceAccount<'info, TokenAccount>,
    pub payer: Signer<'info>,
…
```

And CPI is done here (`packages/layerzero-v2/solana/programs/endpoint/src/instructions/oapp/send.rs:87`):

```
let seeds: &[&[&[u8]]] =
            &[&[MESSAGE_LIB_SEED, send_library.as_ref
              (), &[ctx.accounts.send_library_info.bump]]];
        let cpi_ctx = CpiContext::new_with_signer(
            ctx.accounts.send_library_program.to_account_info(),
            messagelib_interface::cpi::accounts::Interface {
                endpoint: ctx.accounts.send_library_info.to_account_info(),
            },
            seeds,
        )
        .with_remaining_accounts(ctx.remaining_accounts.to_vec());
```

This way, the only account that is validated is the `endpoint` (which also sign the CPI call). The other accounts in CPI call are supplied by the sender using Anchor's remaining accounts. Because these remaining accounts are not validated to maintain composability between the endpoint and various `msglibraries`, it opens the possibility to impersonate the endpoint account (`send_library_info`). Consequently, it is possible to trick the `endpoint`'s `send` instruction into creating a CPI call to `SendWithLzToken`, with the endpoint account being the same as the payer.

Even if there is such context in Anchor:

```
pub endpoint: Signer<'info>,
pub payer: Signer<'info>,
```

If the endpoint and payer is the same account, one signature is enough to sign a transaction.

It is not a severe issue, as the endpoint PDA is not holding lamports and tokens. But still it can break process integrity - we can create a token account for endpoint PDA and send tokens to this account, then supply this account in the endpoint send instruction - so system participants could be tricked into thinking that tokens were sent from an account owned by the endpoint.

In such scenarios:

1. Sending native lamports would fail because only the owner can transfer lamports from the PDA account.
2. Sending tokens could succeed, as the token transfer could occur from an account owned by the `messagelib` PDA.
3. This vulnerability could be exploited to execute malicious transactions, making it appear as though the endpoint (PDA) pays for the transaction.

Corresponding Exploitation Scenarios:

1. A user creates a token account for the PDA.
2. The user sends tokens to the PDA token account.
3. The user exploits the impersonation vulnerability to pay using the PDA token account, effectively executing transactions as though the PDA (endpoint) authorized them.

Location send.rs send_with_lz_token.rs

Recommendation

To mitigate this issue, it is recommended to:

1. Create a check in `SendWithLzToken` to make sure that endpoint and payer are different accounts.
2. Update the documentation about creating a custom `messagelib` to include a note about this attack vector and the necessary check.