Zellic

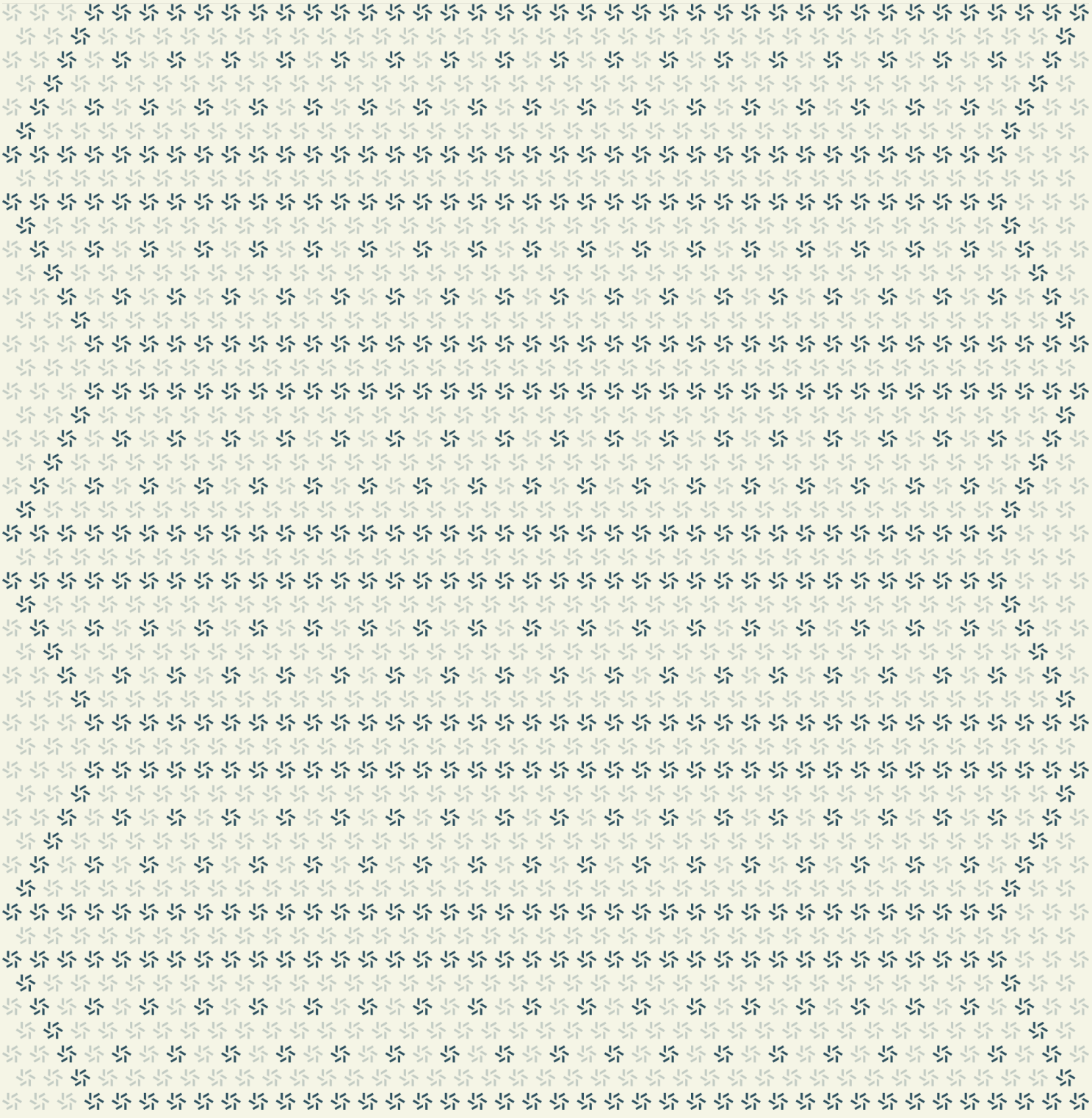
Prepared for
Isaac Zhang
LayerZero Labs

Prepared by
Jasraj Bedi
William Bowling
Zellic

July 23, 2024

LayerZero V2 Solana

Solana Application Security Assessment



Contents

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 from June 10th to July 19th, 2024. During this engagement, Zellic reviewed LayerZero V2 Solana'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:

- Can an OApp be blocked?
- Can an OFT be blocked?
- Is it possible to forge arbitrary messages?
- Is the OApp and OFT validation robust?

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 LayerZero V2 Solana programs, we discovered three findings. No critical issues were found. Two findings were of high impact and one was of medium impact.

Additionally, Zellic recorded its notes and observations from the assessment for LayerZero Labs's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	1
Low	0
Informational	0

2. Introduction

2.1. About LayerZero V2 Solana

LayerZero Labs contributed the following description of LayerZero V2 Solana:

The LayerZero Protocol consists of several programs built on the Solana blockchain designed to facilitate the secure movement of data, tokens, and digital assets between different blockchain environments. LayerZero provides Solana Programs that can communicate directly with the equivalent Solidity Contract Libraries deployed on EVM-based chains.

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. ↗) 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:

LayerZero V2 Solana Programs

Type	Rust
Platform	Solana
Target	monorepo
Repository	https://github.com/LayerZero-Labs/monorepo ↗
Version	`a177aae588eaf0fa9401b86ecb8345c6508fef74`

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

2.4. Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Jasraj Bedi
Co-Founder
jazzy@zellic.io ↗

William Bowling
Engineer
vakzz@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 10, 2024	Start of primary review period
July 19, 2024	End of primary review period

3. Detailed Findings

3.1. Arbitary ULN PDAs can be closed

Target	programs/uln/src/instructions/dvn/commit_verification.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

When the commit verification function is called, all the required and optional DVNs are passed to `check_verifiable`:

```
let dvns_size = config.required_dvns.len() + config.optional_dvns.len();

let confirmation_accounts = &ctx.remaining_accounts[0..dvns_size];
require!(
    check_verifiable(
        &config,
        confirmation_accounts,
        &keccak256(&params.packet_header).to_bytes(),
        &params.payload_hash
    )?,
    UlnError::Verifying
);
```

The `check_verifiable` function then checks that all the required DVNs and at least `optional_dvn_threshold` optional DVNs are verified:

```
pub fn check_verifiable(
    config: &UlnConfig,
    accounts: &[AccountInfo],
    header_hash: &[u8; 32],
    payload_hash: &[u8; 32],
) -> Result<bool> {
    // iterate the required DVNs
    if config.required_dvn_count > 0 {
        for (i, dvn) in config.required_dvns.iter().enumerate() {
            if !verified(dvn, &accounts[i], header_hash, payload_hash,
    config.confirmations)? {
                return Ok(false);
            }
        }
```

```
        // returns early if all required DVNs have signed and there are no
    optional DVNs
        if config.optional_dvn_threshold == 0 {
            return Ok(true);
        }
    }

    // then it must require optional validations
    let mut threshold = config.optional_dvn_threshold as usize;
    let optional_acc_offset = config.required_dvns.len();
    for (i, dvn) in config.optional_dvns.iter().enumerate() {
        if verified(
            dvn,
            &accounts[optional_acc_offset + i],
            header_hash,
            payload_hash,
            config.confirmations,
        )? {
            // increment the optional count if the optional DVN has signed
            threshold -= 1;
            if threshold == 0 {
                // early return if the optional threshold has hit
                return Ok(true);
            }
        }
    }

    // return false as a catch-all
    Ok(false)
}
```

Once `endpoint_verify::verify` has been called, all the `confirmation_accounts` are then passed to `reclaim_storage`:

```
fn reclaim_storage(confirmation_accounts: &[AccountInfo], sol_dest:
    &AccountInfo) -> Result<()> {
    // close accounts
    for acc in confirmation_accounts {
        // skip if account already closed
        if *acc.owner == system_program::ID {
            continue;
        }

        // check if the account is writable to reclaim storage
        require!(acc.is_writable, UlnError::InvalidConfirmation);
```

```
        let lamports = acc.lamports();
        sol_dest.add_lamports(lamports)?;
        acc.sub_lamports(lamports)?;

        acc.assign(&system_program::ID);
        acc.realloc(0, false)?;
    }

    Ok(())
}
```

The issue is that `check_verifiable` will return as soon as `optional_dvn_threshold` has been reached, and any remaining accounts in `confirmation_accounts` are never checked. They will then be passed directly to `reclaim_storage` causing them to be closed.

Impact

A malicious user is able to close any PDA created by the ULN program by first supplying `optional_dvn_threshold` verified optional DVNs, then including the accounts to be closed.

Recommendations

Any account that is passed to `reclaim_storage` should be checked to ensure that it is a valid `Confirmations` account.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in PR #1177 ↗.

3.2. OAPP can reassign executor account

Target	programs/executor/src/instructions/execute.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

When an executor invokes the `lz_receive` function for an OApp, the `executor` account can be passed through as a signer to handle any required fees. The balance of the executor is checked before and after to ensure that it does not lose more that it should:

```
pub fn apply(ctx: &mut Context<Execute>, params: &ExecuteParams) -> Result<()>
    {
    let balance_before = ctx.accounts.executor.lamports();
    let program_id = ctx.remaining_accounts[0].key();
    let accounts = ctx
        .remaining_accounts
        .iter()
        .skip(1)
        .map(|acc| acc.to_account_metas(None)[0].clone())
        .collect::<Vec<_>>();
    let data = get_lz_receive_ix_data(&params.lz_receive)?;
    let result = invoke(&Instruction { program_id, accounts, data },
    ctx.remaining_accounts);

    if let Err(e) = result {
        // call lz_receive_alert
        let params = LzReceiveAlertParams {
            receiver: params.receiver,
            src_eid: params.lz_receive.src_eid,
            sender: params.lz_receive.sender,
            nonce: params.lz_receive.nonce,
            guid: params.lz_receive.guid,
            compute_units: params.compute_units,
            value: params.value,
            message: params.lz_receive.message.clone(),
            extra_data: params.lz_receive.extra_data.clone(),
            reason: e.to_string().into_bytes(),
        };

        let cpi_ctx = LzReceiveAlert::construct_context(
```

```
            ctx.accounts.endpoint_program.key(),
            &[
                ctx.accounts.config.to_account_info(), // use the executor
    config as the signer
                ctx.accounts.endpoint_event_authority.to_account_info(),
                ctx.accounts.endpoint_program.to_account_info(),
            ],
        )?;
        endpoint::cpi::lz_receive_alert(
            cpi_ctx.with_signer(&[&[EXECUTOR_CONFIG_SEED,
    &[ctx.accounts.config.bump]]]),
            params,
        )?;
    } else {
        // assert that the executor account does not lose more than the
    expected value
        let balance_after = ctx.accounts.executor.lamports();
        require!(
            balance_before <= balance_after + params.value,
            ExecutorError::InsufficientBalance
        );
    }
    Ok(())
}
```

The issue is that since the `executor` account can be passed as both a signer and writable to `lz_receive`, it is possible for the OApp to use the `&system_instruction::assign` instruction to take change the owner of the account to itself.

Impact

A malicious OApp can take ownership of an executor account, draining its balance and preventing it from being able to execute transactions.

Recommendations

The owner of the executor should be checked to ensure that it does not change after `lz_receive` is invoked.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in PR #1173 ↗.

3.3. Insufficient remaining_accounts for optional DVNs

Target	programs/uln/src/instructions/dvn/commit_verification.rs		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

As part of the commit verification flow, the `dvns_size` calculation is incorrect. The instruction assumes only the optional DVNs that have committed are passed, whereas the `check_verifiable` iterates over all the `optional_dvns` and indexes into the `remaining_accounts`. This will cause an panic as some of those `optional_dvns` may not have committed, and thus their PDA will not exist.

```
impl CommitVerification<'_> {
    pub fn apply(
        ctx: &mut Context<CommitVerification>,
        params: &CommitVerificationParams,
    ) -> Result<()> {

...
        require!(
            packet_v1_codec::dst_eid(&params.packet_header)
    == ctx.accounts.uln.eid,
            UlnError::InvalidEid
        );

        let dvns_size = config.required_dvns.len() +
    config.optional_dvn_threshold as usize;

...
        require!(
            check_verifiable(
                &config,
                confirmation_accounts,
                &keccak256(&params.packet_header).to_bytes(),
                &params.payload_hash
            )?,
            UlnError::Verifying
        );

pub fn check_verifiable(
    config: &UlnConfig,
```

```
    accounts: &[AccountInfo],
    header_hash: &[u8; 32],
    payload_hash: &[u8; 32],
) -> Result<bool> {

...

    let mut threshold = config.optional_dvn_threshold as usize;
    let optional_acc_offset = config.required_dvns.len();
    for (i, dvn) in config.optional_dvns.iter().enumerate() {
        if verified(
            dvn,
            &accounts[optional_acc_offset + i],
            header_hash,
            payload_hash,
            config.confirmations,
        )? {
            // increment the optional count if the optional DVN has signed
            threshold -= 1;
            if threshold == 0 {
                // early return if the optional threshold has hit
                return Ok(true);
            }
        }
    }
```

Impact

An OApp with N optional DVNs with < N threshold won't be able to succesfully commit, and therefore won't be able to bridge/execute messages.

Recommendations

Fix the `dvns_size` calculation to account for all the `optional_dvns`

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in PR #1043 ↗.

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. Non-PDA OApp signer when registering

When registering an OApp with the endpoint program two signers are required, one is `payer` and the other is `oapp`:

```
#[event_cpi]
#[derive(CpiContext, Accounts)]
#[instruction(params: RegisterOAppParams)]
pub struct RegisterOApp<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,
    /// The PDA of the OApp
    pub oapp: Signer<'info>,
    #[account(
        init,
        payer = payer,
        space = 8 + OAppRegistry::INIT_SPACE,
        seeds = [OAPP_SEED, oapp.key.as_ref()],
        bump
    )]
    pub oapp_registry: Account<'info, OAppRegistry>,
    pub system_program: Program<'info, System>,
}
```

The comments mention that the `oapp` signer should be the PDA of the OApp, but there is nothing stopping regular data accounts from being the `oapp` signer. Since a delegate can be set and is used for authentication in most cases, having a non-PDA account does not gain any additional capabilities. It may cause some confusion if the endpoint program is expecting the `oapp` to always be a PDA from the OFT which does not have to be the case.

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Mainnet.

During our assessment on the scoped LayerZero V2 Solana programs, we discovered three findings. No critical issues were found. Two findings were of high impact and one was of medium impact.

5.1. 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.