OtterSec

LayerZero Solana

Security Assessment

June 7th, 2024 — Prepared by OtterSec

Jessica Clendinen jc0f0@osec.io

Robert Chen notdeghost@osec.io

Table of Contents

01 — Executive Summary

Overview

LayerZero Labs engaged OtterSec to assess the `layerzero-v2` program. This assessment was conducted between May 20th and May 31st, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 7 findings throughout this audit engagement.

In particular, we identified multiple high-risk vulnerabilities related to fee calculation, including one in the pre fee calculation function, which unconditionally returns a pre-fee amount of zero when the fee rate is at its maximum value (OS-LZS-ADV-00), and another issue concerning the failure to differentiate between native and adapter operations, charging transfer fee for native operations like minting and burning (OS-LZS-ADV-01). Furthermore, we highlighted the lack of checks on intermediate accounts passed via remaining accounts, which may result in the sender being spoofed (OS-LZS-ADV-02).

We also made recommendations around improving the efficiency of the system (OS-LZS-SUG-00) and around the need for adherence to coding best practices (OS-LZS-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/LayerZero-Labs/monorepo. This audit was performed against commit fa653ef.

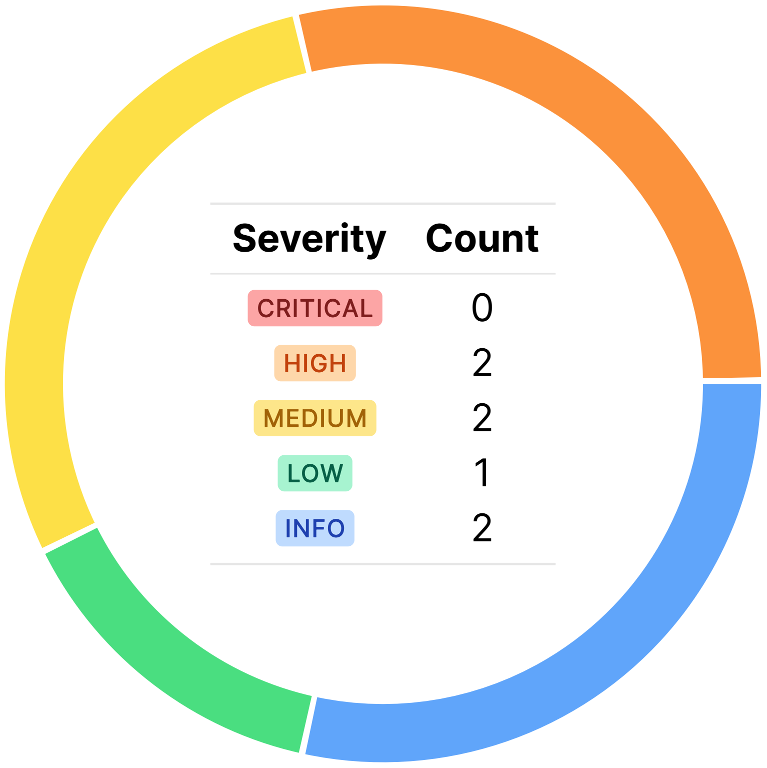
A brief description of the programs is as follows:

Name	Description
layerzero-v2	Implementation of the messaging protocol designed to facilitate the creation of omnichain, interoperable applications on Solana.

03 — Findings

Overall, we reported 7 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	2
MEDIUM	2
LOW	1
INFO	2

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-LZS-ADV-00	HIGH	RESOLVED ✓	`calculate_pre_fee_amount` unconditionally returns a pre-fee amount of zero when the fee rate is at its maximum value.
OS-LZS-ADV-01	HIGH	RESOLVED ✓	`get_pre_fee_amount_ld` fails to differentiate between native and adapter operations for `OFT`, charging fee for minting and burning.
OS-LZS-ADV-02	MEDIUM	RESOLVED ✓	There is a lack of checks on intermediate accounts passed via `remaining_accounts`, which may result in the sender being spoofed.
OS-LZS-ADV-03	MEDIUM	RESOLVED ✓	The fee calculation logic in `quote_treasury` may need clarification, as it always appears to return `treasury_fee`.
OS-LZS-ADV-04	LOW	RESOLVED ✓	`CommitVerification` may pass a signed account to `endpoint_verify` unintentionally via `remaining_accounts`.

Improper Calculation Of Fees HIGH OS-LZS-ADV-00

Description

`quote::get_pre_fee_amount_ld` invokes `calculate_pre_fee_amount` to calculate `amount_sent_ld`. However, there is an issue within `calculate_pre_fee_amount` when `transfer_fee_basis_points` is equal to `MAX_FEE_BASIS_POINTS` which represents a 100% fee. In such a case, the function consistently returns zero as the pre-fee amount.

>_ program-2022/src/extension/transfer_fee/mod.rs rust

```rust
pub fn calculate_pre_fee_amount(&self, post_fee_amount: u64) -> Option<u64> {
    let maximum_fee = u64::from(self.maximum_fee);
    let transfer_fee_basis_points = u16::from(self.transfer_fee_basis_points) as u128;
    if transfer_fee_basis_points == 0 {
        Some(post_fee_amount)
    } else if transfer_fee_basis_points == ONE_IN_BASIS_POINTS || post_fee_amount == 0 {
        Some(0)
    } [...]
}
```

This is problematic because, under a 100% fee rate, the correct pre-fee amount should account for the fact that the entire post-fee amount would be consumed up by the fee.

Remediation

Back-port to PR#6704, which fixes this issue by returning the the pre-fee amount as `post_fee_amount + maximum_fee`, when `transfer_fee_basis_points == MAX_FEE_BASIS_POINTS`.

Patch

Fixed in a555f16.

Charging Fee For Native Operations HIGH OS-LZS-ADV-01

Description

The `quote::get_pre_fee_amount_ld` function calculates the pre-fee amount needed to obtain a specified post-fee amount, taking into account any transfer fees set up in the `TransferFeeConfig` extension. However, this function does not differentiate between various types of `OFT` operations, such as adapter-based transactions and native operations like minting and burning.

>_ *oft/src/instructions/quote.rs* rust

```rust
pub fn get_pre_fee_amount_ld(
    token_mint: &InterfaceAccount<anchor_spl::token_interface::Mint>,
    amount_ld: u64,
) -> Result<u64> {
    let token_mint_info = token_mint.to_account_info();
    let token_mint_data = token_mint_info.try_borrow_data()?;
    let token_mint_unpacked = StateWithExtensions::<Mint>::unpack(&token_mint_data)?;
    Ok(if let Ok(transfer_fee) = token_mint_unpacked.get_extension::<TransferFeeConfig>() {
        transfer_fee
            .get_epoch_fee(Clock::get()?.epoch)
            .calculate_pre_fee_amount(amount_ld)
            .ok_or(ProgramError::InvalidArgument)?
    } else {
        amount_ld
    })
}
```

However, it applies the transfer fee calculation universally, without accounting for the specific context of the operation. For instance, in the case of native operations such as minting and burning, applying transfer fees is not appropriate. These operations are internal to the token's lifecycle and should not be subject to the same transfer fees that external transfers incur.

Remediation

Refactor the functionality to properly distinguish between adapter-based and native operations.

Patch

Fixed in 4aff686.

Insufficient Account Checks MEDIUM OS-LZS-ADV-02

Description

The codebase contains a broad class of vulnerabilities related to insufficient validation of intermediate accounts passed via `remaining_accounts`. These vulnerabilities may enable malicious actors to spoof the sender or manipulate other critical account details, leading to unauthorized actions or token transfers. This issue is particularly pertinent in the context of the `OFT` (Omnichain Fungible Token) `CPI` (Cross-Program Invocation) within the `endpoint` for sending transactions.

The `send::apply` function uses `CPI` to interact with the `endpoint` program, providing transaction details such as the destination endpoint, receiver, message, and associated fees. It also passes `remaining_accounts`. Without proper validation of these accounts, an attacker could insert a spoofed sender account, making it appear as though a legitimate user is initiating the transaction. In reality, the attacker would be in control, potentially leading to unauthorized transfers or actions.

Remediation

Validate that each account in `remaining_accounts` corresponds to the expected signer address.

Patch

Fixed in 101ae40.

Discrepancy In Fee Capping Logic MEDIUM OS-LZS-ADV-03

Description

The `quote::quote_treasury` function begins by calculating `max_native_fee`, which is determined as the greater value between the `worker_fee` and the `treasury_fee_cap`. Next, it computes the `treasury_fee` by applying the treasury's native fee basis points to the `worker_fee` and dividing the result by the denominator (`BPS_DENOMINATOR`). Finally, the function returns the lesser of the two values, `treasury_fee` and `max_native_fee`, as the final native fee for the transaction.

>_ *instructions/endpoint/quote.rs* rust

```rust
pub(crate) fn quote_treasury(
    [...]
) -> Result<u64> {
    if pay_in_lz_token {
        let treasury = treasury.lz_token.as_ref().ok_or(UlnError::LzTokenUnavailable)?;
        Ok(treasury.fee)
    } else {
        // pay in native
        // we must prevent high-treasuryFee Dos attack
        // nativeFee = min(treasuryFee, maxNativeFee)
        // opportunistically raise the maxNativeFee to be the same as _totalNativeFee
        // can't use the _totalNativeFee alone because the oapp can use custom
        // maxNativeFee = max(_totalNativeFee, treasuryNativeFeeCap)
        let max_native_fee = std::cmp::max(worker_fee, treasury_fee_cap);

        let treasury_fee = worker_fee * treasury.native_fee_bps / BPS_DENOMINATOR;

        // nativeFee = min(treasuryFee, maxNativeFee)
        Ok(std::cmp::min(treasury_fee, max_native_fee))
    }
}
```

The function is intended to cap the native fee at `treasury_fee` if it is lower than `max_native_fee`, effectively preventing excessive fees. However, the current implementation returns `treasury_fee` unconditionally. To achieve the intended fee cap, the initial `max` operation should be replaced with a `min` operation.

Remediation

Replace the first `max` operation with a `min`, if the goal is to cap the native fee at the lower of `treasury_fee` and `max_native_fee`.

Patch

Fixed in a555f16.

Potential Signer Leakage LOW

OS-LZS-ADV-04

Description

The `CommitVerification::apply` function invokes `endpoint_verify`, passing `remaining_accounts` as a parameter, along with the seeds used for signing the transaction: `([ULN_SEED, &[ctx.accounts.uln.bu` . If a signed account is inadvertently passed to `endpoint_verify`, the program associated with that account could be executed, potentially leading to unintended consequences.

```rust
>_ uln/src/instructions/dvn/commit_verification.rs                                    rust
pub fn apply(
    ctx: &mut Context<CommitVerification>,
    params: &CommitVerificationParams,
) -> Result<()> {
    [...]
    endpoint_verify::verify(
        ctx.accounts.uln.endpoint_program,
        &params.packet_header,
        params.payload_hash,
        &[ULN_SEED, &[ctx.accounts.uln.bump]],
        &ctx.remaining_accounts[dvns_size..],
    )?;
}
```

Remediation

Modify the code to explicitly validate all the accounts within `remaining_accounts` are marked as read-only and not signed before passing them to `endpoint_verify`.

Patch

Fixed in 5f426f2.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-LZS-SUG-00	Solana explicitly prevents reentrancy, rendering `send_context` unnecessary if `send_library_program` is ensured to be different from `endpoint`.
OS-LZS-SUG-01	Suggestions to ensure adherence to coding best practices.

Code Optimization

OS-LZS-SUG-00

Description

Presently, in `apply` within `send`, `send_context` is utilized to store temporary state information about the `send` operation. This context is utilized to prevent reentrancy, ensuring that no two send operations interfere with each other. However, Solana runtime already prevents reentrancy at the transaction level. Thus, ensuring that the `send_library_program` is not the same as `endpoint` is sufficient to prevent reentrancy.

>_ *endpoint/src/instructions/oapp/send.rs* rust

```rust
pub fn apply<'c: 'info, 'info>(
    ctx: &mut Context<'_, '_, 'c, 'info, Send<'info>>,
    params: &SendParams,
) -> Result<MessagingReceipt> {
    let sender = ctx.accounts.sender.key();
    ctx.accounts.endpoint.send_context = Some(SendContext { dst_eid: params.dst_eid, sender });
    ctx.accounts.endpoint.exit(&ID)?; // store the context into the account
    [...]
}
```

Remediation

Remove `send_context` and instead add a check to ensure that `send_library_program` is different from `endpoint`.

Patch

Fixed in a555f16.

Code Maturity OS-LZS-SUG-01

Description

1. `send_library_config` contains configuration data essential for the message library. Currently, it is not explicitly asserted that this account is rent-exempt. Thus, if `CPI` in `send` invokes a malicious `send_library_program` it may call `close_account`, deleting `send_library_config`.
2. It should be ensured that all destination token accounts are Associated Token Accounts (ATA). Specifically, in `send_with_lz_token` the destination token account for the `LZ` token fee is `message_lib_lz_token` which is not an ATA. Without the guarantee of an ATA, someone may create a token account potentially intercepting the tokens.
3. `burn` in `OftConfigExt::Native` within `send` utilizes `signer` seeds (`with_signer(&[&seeds])`). However, burning tokens does not require explicit authorization from a `signer`. It is a unilateral operation that only depends on the authority granted to the account from which the tokens are being burned.

>_ *oft/src/instructions/send.rs* rust

```rust
pub fn apply(ctx: &mut Context<Send>, params: &SendParams) -> Result<MessagingReceipt> {
    [...]
    match &ctx.accounts.oft_config.ext {
        [...]
        OftConfigExt::Native(_) => {
            // burn
            let cpi_accounts = Burn {
                mint: ctx.accounts.token_mint.to_account_info(),
                from: ctx.accounts.token_source.to_account_info(),
                authority: ctx.accounts.signer.to_account_info(),
            };
            [...]
            token_interface::burn(cpi_context.with_signer(&[&seeds]), amount_sent_ld)?;
        },
    };
    [...]
}
```

Remediation

1. Assert that `send_library_config` is rent-exempt after performing the `CPI`.
2. Ensure that the `message_lib_lz_token` account is an Associated Token Account.
3. The burning operation should be conducted without adding `signer` seeds to the `CPI` context.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL	Vulnerabilities that immediately result in a loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation. • Improperly designed economic incentives leading to loss of funds.
HIGH	Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions. • Exploitation involving high capital requirement with respect to payout.
MEDIUM	Vulnerabilities that may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
LOW	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions.
INFO	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants. • Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.