Solana OFT

Security Assessment

September 25th, 2024 — Prepared by OtterSec

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Layerzero engaged OtterSec to assess the `solana-oft-v2` program. This assessment was conducted between September 6th and September 5th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 3 findings throughout this audit engagement.

In particular, we identified a vulnerability where the fee computation functionality incorrectly removes dust from the sent amount, effectively decreasing the intended value of the sent amount (OS-SFT-ADV-00).

We also recommended utilizing saturating math to prevent any possibility of overflows (OS-SFT-SUG-00) and advised reconsidering the current implementation of seed utilization and future-proofing the burning process for proper execution of transfer hooks (OS-SFT-SUG-01).

Scope

The source code was delivered to us in a Git repository at https://github.com/LayerZero-Labs/monorepo. This audit was performed against a626b15.

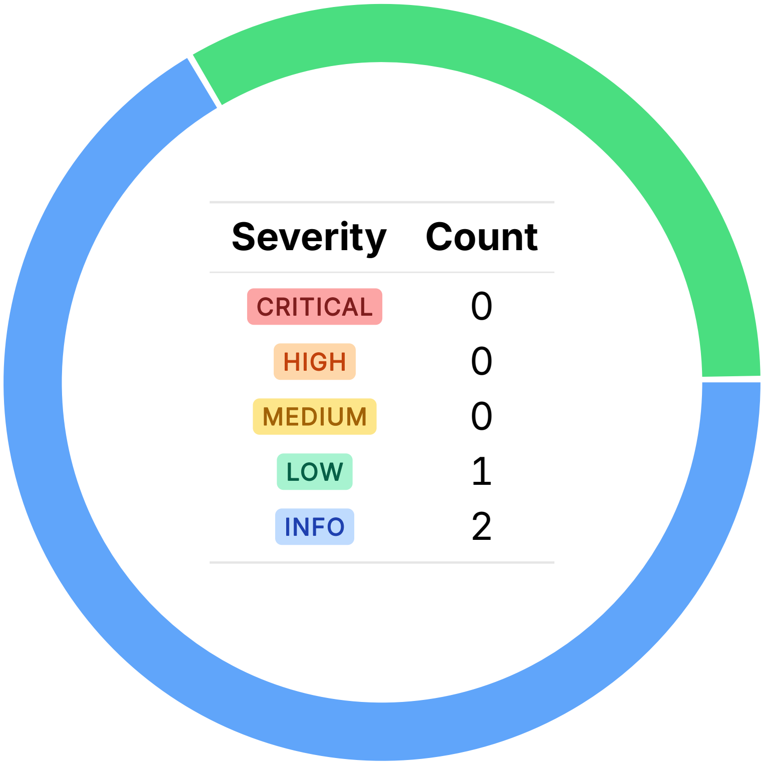
A brief description of the programs is as follows:

Name	Description
solana-oft-v2	It implements the standard for cross-chain token transfers of OFTs in Solana, handling both token and message transfers.

02 — Findings

Overall, we reported 3 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	0
MEDIUM	0
LOW	1
INFO	2

03 — 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-SFT-ADV-00	LOW	RESOLVED ✓	`compute_fee_and_adjust_amount` incorrectly applies `remove_dust` to the `amount_sent_ld`, potentially resulting in the sent amount being less than required.

Incorrect Dust Removal on Sent Amount LOW OS-SFT-ADV-00

Description

In `quote_send::compute_fee_and_adjust_amount`, applying `remove_dust` to the `amount_sent_ld` may result in a situation where the amount sent is slightly less than intended. The issue arises because `remove_dust` is meant to handle very small values that may result in rounding errors when subtracted from `amount_sent_ld`. Thus, removing dust from the sent amount will result in a slight reduction in what is sent.

>_ *oft/src/instructions/quote_send.rs* RUST

```rust
pub fn compute_fee_and_adjust_amount(
    amount_ld: u64,
    oft_store: &OFTStore,
    token_mint: &InterfaceAccount<Mint>,
    fee_bps: Option<u16>,
) -> Result<(u64, u64, u64)> {
    let (amount_sent_ld, amount_received_ld, oft_fee_ld) = if OFTType::Adapter ==
        ↪ oft_store.oft_type
    {
        let mut amount_received_ld =
            oft_store.remove_dust(get_post_fee_amount_ld(token_mint, amount_ld)?);
        let amount_sent_ld =
            oft_store.remove_dust(get_pre_fee_amount_ld(token_mint, amount_received_ld)?);
        [...]
        amount_received_ld -= oft_fee_ld;
        (amount_sent_ld, amount_received_ld, oft_fee_ld)
    } else {
        // if it is Native OFT, there is no transfer fee
        let amount_sent_ld = oft_store.remove_dust(amount_ld);
        [...]
        let amount_received_ld = amount_sent_ld - oft_fee_ld;
        (amount_sent_ld, amount_received_ld, oft_fee_ld)
    };
    Ok((amount_sent_ld, amount_received_ld, oft_fee_ld))
}
```

Remediation

Ensure `amount_sent_ld` remains untouched such that the full intended amount is sent.

Patch

Resolved in PR#1767.

04 — 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-SFT-SUG-00	Suggestion to utilize saturating math to prevent any possibility of overflows in `RateLimiter::refill`.
OS-SFT-SUG-01	Recommendations for modifying the codebase to improve efficiency and mitigate potential security issues.

Overflow Mitigation

OS-SFT-SUG-00

Description

`RateLimiter::refill` in `peer_config` should utilize saturating math (`saturating_add` and `saturating_mul`) to prevent potential overflows or incorrect calculations when adding tokens, especially when large values are involved. Saturating math ensures that when operations exceed the maximum value that may be stored in a type, they cap at the maximum value instead of overflowing.

>_ *oft/src/state/peer_config.rs Latest* RUST

```rust
pub fn refill(&mut self, extra_tokens: u64) -> Result<()> {
    let mut new_tokens = extra_tokens;
    let current_time: u64 = Clock::get()?.unix_timestamp.try_into().unwrap();
    if current_time > self.last_refill_time {
        let time_elapsed_in_seconds = current_time - self.last_refill_time;
        new_tokens += time_elapsed_in_seconds * self.refill_per_second;
    }
    self.tokens = std::cmp::min(self.capacity, self.tokens.saturating_add(new_tokens));
    self.last_refill_time = current_time;
    Ok(())
}
```

Remediation

Implement the above-mentioned suggestions.

Patch

Resolved in PR#1767.

Code Refactoring OS-SFT-SUG-01

Description

1. In multiple places in the current implementation of the **OFT** instructions, the seeds do not serve any meaningful purpose and result in unnecessary costs. Thus, it would be cost-effective to remove them entirely for improved optimization.

   >_ *oft/src/instructions/...* RUST

   ```rust
   #[account(
       seeds = [OFT_SEED, oft_store.token_escrow.as_ref()],
       bump = oft_store.bump,
       has_one = admin @OFTError::Unauthorized
   )]
   pub oft_store: Account<'info, OFTStore>,
   ```

2. It would be appropriate to transfer tokens to a shared vault and then burn them, rather than directly burning tokens from a user's token account as done presently. Although this may not be immediately necessary, it will help with future compatibility, ensuring any transfer hooks are properly invoked in the future.

Remediation

Incorporate the above-stated modifications into the codebase.

Patch

Resolved in PR#1767.

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 others may have 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.