Layer Zero Security Review

Pashov Audit Group

Conducted by: Koolex, castle_chain, ubermensch

September 12th - September 17th

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/devtools/examples/oft-solana** 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 OFT 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 - 6a07bb7e089349995ce4b30978512aff001cc131

Scope

The following smart contracts were in scope of the audit:

- `init_oft`
- `lz_receive`
- `lz_receive_types`
- `mod`
- `quote_oft`
- `quote_send`
- `send`
- `set_oft_config`
- `set_pause`
- `set_peer_config`
- `withdraw_fee.rs`
- `oft`
- `peer_config`
- `compose_msg_codec`
- `errors`
- `events`
- `lib`
- `msg_codec`

7. Executive Summary

Over the course of the security review, Koolex, castle_chain, ubermensch engaged with Layer Zero to review Layer Zero. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Layer Zero
Repository	https://github.com/LayerZero-Labs/devtools
Date	September 12th - September 17th
Protocol Type	Messaging protocol

Findings Count

Severity	**Amount**
Medium	1
Low	11
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Mint decimal manipulation	Medium	Acknowledged
[L-01]	Lack of message size validation in OFT encoding	Low	Acknowledged
[L-02]	Inconsistent compose_msg implementation	Low	Acknowledged
[L-03]	Lack of flexible account sizing and data reallocation for future upgrades	Low	Acknowledged
[L-04]	Lack of validation for peer address	Low	Acknowledged
[L-05]	Lack of minimum threshold check on fee withdrawal	Low	Acknowledged
[L-06]	Rate Limiter vulnerability to large transaction dominance	Low	Acknowledged
[L-07]	Critical OFT settings vulnerable to Solana restarts	Low	Acknowledged
[L-08]	DOS risk due to frozen token_escrow	Low	Acknowledged
[L-09]	Unlimited decimals of local token mints	Low	Acknowledged
[L-10]	Multiple native OFTs can be created for the same token mint	Low	Acknowledged
[L-11]	Users can avoid paying fees	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Mint decimal manipulation

Severity

Impact: High

Likelihood: Low

Description

The `ld2sd_rate` (local-to-shared decimal rate) can be manipulated by the initializer through the exploitation of the `MintCloseAuthority` extension in the Solana program. This manipulation is possible because the initializer has control over the mint's decimal value, which can be changed after the mint's creation, leading to critical discrepancies in token accounting and potential financial exploits.

The process to execute this exploit is as follows:

1. The admin creates a mint with an initial decimal value of 18 using the `MintCloseAuthority` extension, assigning the close authority to an address they control.
2. The admin uses the mint in the `InitOFT` resulting in 1e12 as a value of the `ld2sd_rate` (assuming 6 shared decimals).
3. With the supply of the mint still at 0, the admin then uses the close authority to close the mint account.
4. After the mint is closed, the admin can reinitialize a new mint at the same address, but this time with a reduced decimal value, such as 6.

This manipulation of decimal values causes the `ld2sd_rate` to become inflated, as the program (OFT) will still treat the mint as though it has 18

decimals while it actually operates with only 6 decimals. This mismatch leads to erroneous token calculations and can be used for various financial exploits.

Example Exploits:

- **Legitimate transfers treated as dust:**
  Assume a token with a value of $1, where 1 token equals `1e6` units (decimal of 6). An admin or attacker can send 100,000 tokens (with a total value of $1,000,000). Using the manipulated `ld2sd_rate`, the program with an inflated rate of `1e12`, causing the whole amount `1e11` (100 billion units) as a dust due to `remove_dust`, meaning the tokens will not be sent as intended.

- **Cross-chain Manipulation:**
  The attacker can initialize another `OFTStore` with a different token escrow account, using the manipulated `ld2sd_rate` to transfer tokens from another chain (e.g., Ethereum) to Solana. The inflated rate on Solana causes the amount received to be much higher than intended. Once the tokens are transferred, the attacker switches the peer to the new `OFTStore` using the correct (lower) rate and transfers the tokens back to the original chain, gaining an arbitrage-like advantage due to the discrepancy in the rates between the `OFTStore` accounts.

```rust
use crate::*;
use anchor_spl::token_interface::{Mint, TokenAccount, TokenInterface};
use oapp::endpoint::{instructions::RegisterOAppParams, ID as ENDPOINT_ID};

#[derive(Accounts)]
pub struct InitOFT<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,
    #[account(
        init,
        payer = payer,
        space = 8 + OFTStore::INIT_SPACE,
        seeds = [OFT_SEED, token_escrow.key().as_ref()],
        bump
    )]
    pub oft_store: Account<'info, OFTStore>,
    #[account(
        init,
        payer = payer,
        space = 8 + LzReceiveTypesAccounts::INIT_SPACE,
        seeds = [LZ_RECEIVE_TYPES_SEED, oft_store.key().as_ref()],
        bump
    )]
    pub lz_receive_types_accounts: Account<'info, LzReceiveTypesAccounts>,
    #[account(mint::token_program = token_program)]
    pub token_mint: InterfaceAccount<'info, Mint>,
    #[account(
        init,
        payer = payer,
        token::authority = oft_store,
        token::mint = token_mint,
        token::token_program = token_program,
    )]
    pub token_escrow: InterfaceAccount<'info, TokenAccount>,
    pub token_program: Interface<'info, TokenInterface>,
    pub system_program: Program<'info, System>,
}

impl InitOFT<'_> {
    pub fn apply(ctx: &mut Context<InitOFT>, params: &InitOFTParams) -> Result<
      ()> {
        // Initialize the oft_store
        ctx.accounts.oft_store.oft_type = params.oft_type.clone();
        require!(
            ctx.accounts.token_mint.decimals >= params.shared_decimals,
            OFTError::InvalidDecimals
        );
    }
```

The current implementation does not validate or ensure that the `MintCloseAuthority` is disabled. An attacker can reinitialize a mint account after closing the previous one, as the rate does not adapt after initialization.

Recommendations

To mitigate this issue, it is strongly recommended to:

1. Add a check in the `InitOFT` instruction to verify that the `MintCloseAuthority` extension is not enabled.

2. Ensure that the close authority for the mint is explicitly set to `None` during initialization.

This will prevent the exploitation of the `MintCloseAuthority` and ensure that the mint's decimal value cannot be manipulated after its creation, thereby safeguarding the `ld2sd_rate` from being inflated.

8.2. Low Findings

[L-01] Lack of message size validation in OFT encoding

The OFT implementation lacks explicit size checks for the `compose_msg` parameter in its encoding functions. This could potentially lead to the creation of oversized messages, which may cause issues in cross-chain communication and Solana transaction processing.

The affected code is primarily in the `msg_codec.rs` and `compose_msg_codec.rs` files:

```rust
// In msg_codec.rs
pub fn encode(
    send_to: [u8; 32],
    amount_sd: u64,
    sender: Pubkey,
    compose_msg: &Option<Vec<u8>>,
) -> Vec<u8> {
    if let Some(msg) = compose_msg {
        let mut encoded = Vec::with_capacity(72 + msg.len()); // 32 + 8 + 32
        encoded.extend_from_slice(&send_to);
        encoded.extend_from_slice(&amount_sd.to_be_bytes());
        encoded.extend_from_slice(sender.to_bytes().as_ref());
        encoded.extend_from_slice(&msg);
        encoded
    } else {
        // ... (omitted for brevity)
    }
}

// In compose_msg_codec.rs
pub fn encode(
    nonce: u64,
    src_eid: u32,
    amount_ld: u64,
    compose_msg: &Vec<u8>, // [composeFrom][composeMsg]
) -> Vec<u8> {
    let mut encoded = Vec::with_capacity(20 + compose_msg.len()); // 8 + 4 + 8
    encoded.extend_from_slice(&nonce.to_be_bytes());
    encoded.extend_from_slice(&src_eid.to_be_bytes());
    encoded.extend_from_slice(&amount_ld.to_be_bytes());
    encoded.extend_from_slice(&compose_msg);
    encoded
}
```

[L-02] Inconsistent `compose_msg` implementation

OFT implementation contains two different versions of the `compose_msg` function, located in `msg_codec.rs` and `compose_msg_codec.rs`. These functions handle the extraction of the composed message from the input byte array. The inconsistency between these implementations could potentially lead to silent failures and unexpected behavior in certain scenarios.

In `msg_codec.rs`, the implementation is as follows:

```
pub fn compose_msg(message: &[u8]) -> Option<Vec<u8>> {
    if message.len() > COMPOSE_MSG_OFFSET {
        Some(message[COMPOSE_MSG_OFFSET..].to_vec())
    } else {
        None
    }
}
```

This implementation safely returns an `Option<Vec<u8>>`, allowing the caller to handle cases where the message is too short explicitly.

However, in `compose_msg_codec.rs`, the implementation is:

```
pub fn compose_msg(message: &[u8]) -> Vec<u8> {
    if message.len() > COMPOSE_MSG_OFFSET {
        message[COMPOSE_MSG_OFFSET..].to_vec()
    } else {
        Vec::new()
    }
}
```

This version returns an empty `Vec` when the message is too short, potentially leading to silent failures if not handled properly by the caller.

The impact of this inconsistency is mitigated by the fact that the main program logic primarily uses the safer implementation from `msg_codec.rs`. However, the existence of the potentially problematic implementation in `compose_msg_codec.rs` introduces a risk of unexpected behavior, particularly if future modifications inadvertently use this version.

Standardize the `compose_msg` function implementation across the codebase. Replace the implementation in `compose_msg_codec.rs` with the safer version that returns `Option<Vec<u8>>`:

[L-03] Lack of flexible account sizing and data reallocation for future upgrades

Description

The protocol currently uses hardcoded account sizes without proper checks to ensure accounts have sufficient space for future updates. It also does not handle potential account data reallocation, which Solana supports. This lack of flexibility may complicate future upgrades to account structures.

For example, in the `init_oft` function:

```
#[account(mut)]
    pub payer: Signer<'info>,
    #[account(
        init,
        payer = payer,
        space = 8 + OFTStore::INIT_SPACE,
        seeds = [OFT_SEED, token_escrow.key().as_ref()],
        bump
    )]
    pub oft_store: Account<'info, OFTStore>,
```

Using a constant space for the account will make it difficult to modify or expand the account in future updates.

Recommendation

Add padding space to the Program Derived Accounts (PDAs) to allow for future protocol updates. This will give the protocol the flexibility to handle changes without the need for complex migrations or reallocations later on.

For instance, you can increase the space allocation like this:

```
-         space = 8 + OFTStore::INIT_SPACE,
+         space = 8 + OFTStore::INIT_SPACE + PADDING_SPACE,
```

Where `PADDING_SPACE` is an additional buffer added to accommodate future fields or changes in the account's structure.

[L-04] Lack of validation for peer address

The OFT lacks proper validation of the peer address during cross-chain token transfers. Specifically, in the `send` function, the `peer_address` is used directly from the peer configuration without verifying its correspondence to the destination endpoint ID (`dst_eid`).

```
let msg_receipt = oapp::endpoint_cpi::send(
    ctx.accounts.oft_store.endpoint_program,
    ctx.accounts.oft_store.key(),
    ctx.remaining_accounts,
    &[OFT_SEED, ctx.accounts.token_escrow.key().as_ref
      (), &[ctx.accounts.oft_store.bump]],
    EndpointSendParams {
        dst_eid: params.dst_eid,
        receiver: ctx.accounts.peer.peer_address,
        message: msg_codec::encode(
            params.to,
            amount_sd,
            ctx.accounts.signer.key(),
            &params.compose_msg,
        ),
        // ... other parameters ...
    },
)?;
```

While the `peer` account is loaded based on the `dst_eid`, there's no explicit check to confirm that the stored `peer_address` is indeed the correct one for the given `dst_eid`.

Implement a validation mechanism in the `send` function to verify the correctness of the `peer_address` for the given `dst_eid` before initiating the transfer.

[L-05] Lack of minimum threshold check on fee withdrawal

In the `withdraw_fee` function, while there is a check to ensure that the withdrawn amount doesn't exceed the available fee balance, no minimum threshold is enforced.

```
pub fn apply
  (ctx: &mut Context<WithdrawFee>, params: &WithdrawFeeParams) -> Result<()> {
    require!(

                ctx.accounts.token_escrow.amount - ctx.accounts.oft_store.tvl_ld >= p
        OFTError::InvalidFee
    );
    // ... rest of the function
}
```

In case, a fee-on-transfer is taken, the admin might receive zero (or too low) amount after deducting the fee. This is likely when the fee withdrawal is automated by a program or a client-side bot.

Consider adding a slippage protection or enforcing a minimum threshold.

[L-06] Rate Limiter vulnerability to large transaction dominance

The rate limiter is vulnerable to domination by large transactions, particularly in Solana's high-throughput environment. The rate limiter operates on a first-come, first-served basis without considering transaction size or user fairness, which can lead to rapid depletion of the rate limit and unfair resource allocation.

Affected code:

```rust
// In peer_config.rs
pub fn try_consume(&mut self, amount: u64) -> Result<()> {
    self.refill(0)?;
    match self.tokens.checked_sub(amount) {
        Some(new_tokens) => {
            self.tokens = new_tokens;
            Ok(())
        },
        None => Err(error!(OFTError::RateLimitExceeded)),
    }
}

// In send.rs
if let Some(rate_limiter) = ctx.accounts.peer.outbound_rate_limiter.as_mut() {
    rate_limiter.try_consume(amount_received_ld)?;
}
```

Potential Flow:

1. A user initiates a large transaction (i.e. high amount) that consumes a significant portion of the rate limit.
2. Due to Solana's high throughput, multiple users may attempt similar large transactions almost simultaneously.
3. The first transaction processed depletes most or all of the rate limit.
4. Subsequent transactions, regardless of size, fail due to insufficient rate limit tokens.
5. The rate limit refills over time, but the cycle can repeat with the next large transaction.

Potential Impact:

- Large transactions can monopolize the bridge's capacity, preventing smaller transactions from being processed.
- Potential Denial of Service: A series of large transactions could consistently deplete the rate limit, effectively denying service to other users.

Implement a token bucket algorithm with separate buckets for different transaction size ranges (amount). This would ensure that both large and small transactions have allocated capacity, preventing large transactions from dominating the entire rate limit.

[L-07] Critical OFT settings vulnerable to Solana restarts

The OFTStore structure contains critical settings that are vulnerable to Solana chain restarts. Specifically, the `paused` and `default_fee_bps` fields can revert to previous states.

```
pub struct OFTStore {
    // ... other fields ...
    pub default_fee_bps: u16,
    pub paused: bool,
    // ... other fields ...
}
```

If Solana restarts to a previous slot:

1. The `paused` field could revert from true to false, reactivating the system when it should be paused for security reasons. This is particularly dangerous if the pause was enacted in response to a detected vulnerability or ongoing attack.
2. The `default_fee_bps` could revert to an outdated value. This might lead to incorrect fee calculations, potentially causing financial losses for users or the protocol itself.

These issues arise because Solana's restart mechanism reverts the entire state (when the validators vote for that) to a previous slot, including these critical OFT settings. The current implementation doesn't have safeguards against such cases.

Consider utilizing **LastRestartSlot** sysvar to detect outdated config. If the config is outdated, consider the protocol paused till an action is taken by the admin.

An example of that:

```
pub struct OFTStore {
    // ... other fields ...
    pub last_updated_slot: u64,
    // ... other fields ...
}

fn is_config_outdated(oft_store: &OFTStore) -> Result<bool> {
    let last_restart_slot = LastRestartSlot::get()?;
    Ok(oft_store.last_updated_slot <= last_restart_slot.last_restart_slot)
}
```

[L-08] DOS risk due to frozen `token_escrow`

In the `init_oft` instruction, the `token_mint` is set without validation, allowing the initialization of a `token_mint` with a `freeze_authority`. SPL tokens with a freeze authority can have their accounts frozen by the token issuer or an authorized entity, posing a risk to the functioning of the OFT (Omni-Chain Fungible Token).

For example:

```
pub struct InitOFT<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,
    #[account(
        init,
        payer = payer,
        space = 8 + OFTStore::INIT_SPACE,
        seeds = [OFT_SEED, token_escrow.key().as_ref()],
        bump
    )]
    pub oft_store: Account<'info, OFTStore>,
    #[account(
        init,
        payer = payer,
        space = 8 + LzReceiveTypesAccounts::INIT_SPACE,
        seeds = [LZ_RECEIVE_TYPES_SEED, oft_store.key().as_ref()],
        bump
    )]
    pub lz_receive_types_accounts: Account<'info, LzReceiveTypesAccounts>,
    #[account(mint::token_program = token_program)]
    pub token_mint: InterfaceAccount<'info, Mint>,
}
```

If the `token_escrow` is frozen, it will be impossible to transfer the token fees to it, causing a Denial of Service (DoS) in the `send` instruction. This will render the OFT unusable because fee transfers will revert at this point:

```
if oft_fee_ld > 0 {
    token_interface::transfer_checked(
        CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            TransferChecked {
                from: ctx.accounts.token_source.to_account_info(),
                mint: ctx.accounts.token_mint.to_account_info(),
                to: ctx.accounts.token_escrow.to_account_info(),
                authority: ctx.accounts.signer.to_account_info(),
            },
        ),
        oft_fee_ld,
        ctx.accounts.token_mint.decimals,
    )?;
}
```

1. During OFT initialization, check if the `token_mint` has a `freeze_authority` and return an error if detected.
2. If support for such tokens is necessary, display a warning on the UI to inform traders of the associated risks.
3. Keep in mind that major regulated stablecoins, such as USDC, have a `freeze_authority` for security reasons (e.g., preventing money laundering). If the protocol wishes to support USDC or similar tokens, implement an allowlist for trusted tokens while applying strict checks on others.

[L-09] Unlimited decimals of local token mints

In the `init_oft()` function:

```
require!(
            ctx.accounts.token_mint.decimals >= params.shared_decimals,
            OFTError::InvalidDecimals
        );
```

This code should cap the maximum decimals to ensure values can be stored in a `u64`, as required by the protocol.

In the `lz_receive()` function, if `ld2sd` has a large value due to a significant decimal difference between the local and shared decimals, it will cause an overflow. Although it will revert, the math used in `sd2ld` is not checked:

```
let mut amount_received_ld = ctx.accounts.oft_store.sd2ld(amount_sd);
```

The `sd2ld` function does not use `checked_mul` to prevent overflow:

```
pub fn sd2ld(&self, amount_sd: u64) -> u64 {
        amount_sd * self.ld2sd_rate
    }
```

The `ld2sd_rate` is calculated here:

```
ctx.accounts.oft_store.ld2sd_rate =
            10u64.pow(
              (ctx.accounts.token_mint.decimals - params.shared_decimals) as u32);
```

Proof of Concept (POC) If the local token has 18 decimals and the shared token has 6 decimals, the `ld2sd_rate` will be `10^(12)`. If we receive an amount of `10_000 * 10^(6)`, the amount calculated from `sd2ld` will be `10_000 * 10^(18)`. This value cannot be stored in a `u64`, leading to an overflow.

Consider adding a decimal limit in the `init_oft` instruction, allowing only tokens with appropriate decimals to be stored in a `u64`.

```
+        require!(
+            ctx.accounts.token_mint.decimals - params.shared_decimals <= 9,
+            OFTError::InvalidDecimals
+        );

+        require!(
+            ctx.accounts.token_mint.decimals <= 12,
+            OFTError::InvalidDecimals
+        );
```

[L-10] Multiple native OFTs can be created for the same token mint

According to the technical docs :

> each token mint can be associated with up to 1 native OFT and unlimited adapter OFTs

The protocol emphasizes that each token mint can be associated with only **one Native OFT** but can have multiple Adapter OFTs. However, this constraint is

not enforced in the `init_oft` instruction.

In the current implementation, the `oft_store` is initialized with a seed based on the `token_escrow`, as shown below:

```
pub struct InitOFT<'info> {
    #[account(
        init,
        payer = payer,
        space = 8 + OFTStore::INIT_SPACE,
        seeds = [OFT_SEED, token_escrow.key().as_ref()],
        bump
    )]
    pub oft_store: Account<'info, OFTStore>,

    #[account(
        init,
        payer = payer,
        token::authority = oft_store,
        token::mint = token_mint,
        token::token_program = token_program,
    )]
    pub token_escrow: InterfaceAccount<'info, TokenAccount>,
```

Since `token_escrow` is used in the seed, it is possible to create multiple `oft_store` accounts for the same `token_mint`, which is permissible for Adapter OFTs but violates the protocol's rule that **only one Native OFT** should exist per token mint. The code does not enforce this rule, leading to the potential creation of multiple Native OFTs for the same token mint, which contradicts the protocol's intended design.

Impact If multiple Native OFTs are created for the same `token_mint`, it could break the protocol's guarantees, resulting in inconsistencies, potential conflicts in token handling, and unexpected behavior in cross-chain operations. This would undermine the protocol's intended structure and could lead to issues in managing token supply and interactions.

Enforce the protocol's rule by making the seed for the `oft_store` conditional based on the type of OFT:

- **Native OFT:** Use the `token_mint` as the seed to guarantee that only one `Native OFT` can exist for each token mint.
- **Adapter OFT:** Use the `token_escrow` as the seed to allow for multiple Adapter OFTs for each token mint.

For example:

```
if oft_type == OFTType::Native {
+    seeds = [OFT_SEED, token_mint.key().as_ref()];
} else {
+    seeds = [OFT_SEED, token_escrow.key().as_ref()];
}
```

This ensures that only one Native OFT is created per token mint while allowing flexibility for multiple Adapter OFTs.

[L-11] Users can avoid paying fees

The vulnerability exists in the `send` function, which is responsible for sending tokens across chains. This function requires the user to pay fees to the protocol in the local representation of the token, as demonstrated in the following code:

```
if oft_fee_ld > 0 {
    token_interface::transfer_checked(
        CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            TransferChecked {
                from: ctx.accounts.token_source.to_account_info(),
                mint: ctx.accounts.token_mint.to_account_info(),
                to: ctx.accounts.token_escrow.to_account_info(),
                authority: ctx.accounts.signer.to_account_info(),
            },
        ),
        oft_fee_ld,
        ctx.accounts.token_mint.decimals,
    )?;
}
```

Although fees are paid in the local token, the protocol mistakenly applies dust removal to the fee amount. Dust removal should only be applied to the amount being transferred cross-chain, as it involves different decimals. Since the fees are paid locally, dust removal on fees is inappropriate and results in potential fee avoidance by users. The fee calculation includes dust removal as shown below:

```
let oft_fee_ld = oft_store.remove_dust(calculate_fee(
    amount_received_ld,
    oft_store.default_fee_bps,
    fee_bps,
));
```

This opens the door to a critical vulnerability where users can manipulate the fee calculation. By sending small amounts, users can make their fees zero due to dust removal, effectively bypassing the protocol's fee system.

The `remove_dust` function is implemented as follows:

```
pub fn remove_dust(&self, amount_ld: u64) -> u64 {
    amount_ld - amount_ld % self.ld2sd_rate
}
```

If the amount sent by a user results in a fee that can be reduced to zero due to dust removal, the protocol will lose fees, even if the `default_fee_bps` is set at a significant value.

Exploitation Scenario

- Local decimals: 9
- Shared decimals: 3 (or 6)
- Amount sent by user: 19,999,999
- `OFTType`: Native
- `default_fee_bps`: 500 (equivalent to 5%)

In this scenario, the amount received after dust removal would be 19,000,000. The fees taken by the protocol would be zero, representing a significant loss of funds.

Proof of Concept

You can add the following test in the `quote_send.rs` file and observe the printed values:

```
#[test]
fn tests() {
    let mock_addr = Pubkey::default();

    let ofts = OFTStore {
        oft_type: OFTType::Native,
        ld2sd_rate: 10u64.pow(6),
        token_mint: mock_addr,
        token_escrow: mock_addr,
        endpoint_program: mock_addr,
        bump: 1 as u8,
        tvl_ld: 100_000_000 as u64,
        admin: mock_addr,
        default_fee_bps: 500 as u16,
        paused: false,
        pauser: Some(mock_addr),
        unpauser: Some(mock_addr),
    };

    let amountld: u64 = 19_999_999;
    let amount_received = ofts.remove_dust(amountld);

    println!(
      "Amountreceivedfromtheuserafterdustremoval{:?}",
      amount_received
    );

    let oft_fee = ofts.remove_dust(calculate_fee
      (amount_received, ofts.default_fee_bps, None));

    println!("The amount of fee taken from the user: {:?}", oft_fee);
}
```

Expected Output

```
Amount received from the user after dust removal: 19,000,000
The amount of fee taken from the user: 0
```

To address this issue, avoid applying `remove_dust` on the fee amount. The fees should reflect their true value without unnecessary rounding. The recommended change is shown below:

```
- let oft_fee_ld = oft_store.remove_dust(calculate_fee(
-     amount_received_ld,
-     oft_store.default_fee_bps,
-     fee_bps,
- ));

+ let oft_fee_ld = calculate_fee(
+     amount_received_ld,
+     oft_store.default_fee_bps,
+     fee_bps,
+ );
```