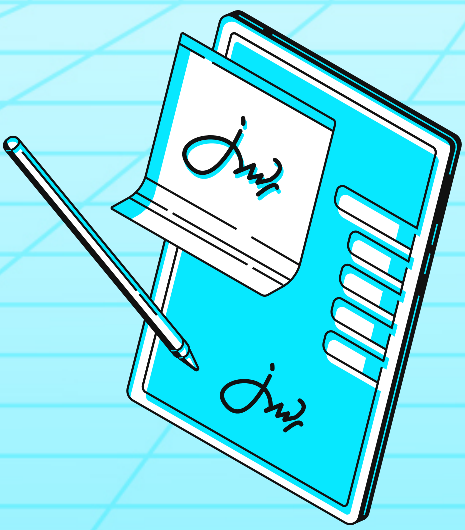
our shielding · Your smart contracts, our shielding · Your smart c

shieldify



LzApp ONFT

SECURITY REVIEW

Date: 25 September 2024

CONTENTS

1. **About Shieldify Security** 3
2. **Disclaimer** 3
3. **About LzApp ONFT** 3
4. **Risk classification** 3
   - 4.1 Impact 3
   - 4.2 Likelihood 3
5. **Security Review Summary** 4
   - 5.1 Protocol Summary 4
   - 5.2 Scope 4
6. **Findings Summary** 5
7. **Findings** 6

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About LzApp ONFT

This project implements an Omnichain Non-Fungible Token (ONFT) on the Solana blockchain using LayerZero's cross-chain messaging protocol. Unlike the Ethereum implementation which uses contract inheritance, this Solana version utilizes LayerZero's Endpoint Cross Program Invocation (CPI) Helper to achieve cross-chain functionality. The Solana program is structured to work with LayerZero's messaging interface, allowing for sending and receiving arbitrary data between different blockchain networks.

4. Risk Classification

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

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible

- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 4 days with a total of 64 hours dedicated by Oxcastle_chain from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying one Critical and one High-severity issue, related to decimals manipulation and denial of service due to missing validation.

The development team has done a great job with their test suite and provided support and responses to all of the questions that the Shieldify researchers had.

5.1 Protocol Summary

Project Name	**LzApp**
Repository	LzApp-ONFT
Type of Project	Omnichain Non-Fungible Token
Audit Timeline	4 days
Review Commit Hash	5118ac635033f3b23d31724661337e7feb7e6559
Fixes Review Commit Hash	39c8ae02da086db1fadbca8c9e6000f0f75f9060

5.2 Scope

The following smart contracts were in the scope of the security review:

File	LOC
/instructions/init_adapter_oft.rs	72
/instructions/init_oft.rs	72
/instructions/lz_receive_types.rs	103
/instructions/lz_receive.rs	153
/instructions/mint_to.rs	48
/instructions/mod.rs	31
/instructions/quote_oft.rs	80
/instructions/quote.rs	152
/instructions/send.rs	162
/instructions/set_delegate.rs	33
/instructions/set_enforced_options.rs	41
/instructions/set_mint_authority.rs	32
/instructions/set_peer.rs	34

/instructions/set_rate_limit.rs	42
/instructions/transfer_admin.rs	23
/state/enforced_options.rs	31
/state/mod.rs	6
/state/onft.rs	76
/state/peer.rs	50
compose_msg_codec.rs	45
errors.rs	18
events.rs	32
lib.rs	99
msg_codec.rs	45
Total	**1480**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical and High** issues: **2**
- **Medium** issues: **3**
- **Low** issues: **3**

ID	Title	Severity	Status
[C-01]	Mint decimal manipulation through `MintCloseAuthority` leads to inflation of `ld2sd_rate`	Critical	Fixed
[H-01]	Denial of Service Risk Due to Frozen `token_escrow` in `ONFT::Adapter`	High	Fixed
[M-01]	Unchecked Math Operations Leading to Potential Arithmetic Errors	Medium	Fixed
[M-02]	Unlimited Decimals of Local Token Mints Result in DoS and Potential Overflow	Medium	Fixed
[M-03]	Missing Size Checks for `compose_msg` can Lead to Oversized Messages and Transaction Failures	Medium	Fixed
[L-01]	Require New Admin as Co-signer for Admin Setting in `transfer_admin` instruction	Low	Fixed
[L-02]	Inconsistent `compose_msg` Function Implementation May Lead to Silent Failures	Low	Fixed
[L-03]	Lack of Pause Functionality for Cross-Chain Token Transfers Could Lead to Security Risks	Low	Acknowledged

7. Findings

[C-01] Mint decimal manipulation through `MintCloseAuthority` leads to inflation of `ld2sd_rate`

Severity

Critical Risk

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 `InitONFT` resulting in 1e12 as a value of the `ld2sd_rate` (assuming 6 shared decimals).
3. With the mint supply 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 (ONFT) 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) to dust due to `remove_dust`, meaning the tokens will not be sent as intended.
- **Cross-chain Manipulation:**
  The attacker can initialize another `ONFT_config` 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 `ONFT_config` 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 `ONFT_config` accounts.

```rust
pub struct InitAdapterONft<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,
    #[account(
        init,
        payer = payer,
        space = 8 + ONftConfig::INIT_SPACE,
        seeds = [ONft_SEED, token_escrow.key().as_ref()],
        bump
    )]
    pub ONft_config: Account<'info, ONftConfig>,
    #[account(
        init,
        payer = payer,
        space = 8 + LzReceiveTypesAccounts::INIT_SPACE,
        seeds = [LZ_RECEIVE_TYPES_SEED, &ONft_config.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 = ONft_config,
        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 InitAdapterONft<_> {
    pub fn apply(ctx: &mut Context<InitAdapterONft>, params: &
        InitAdapterONftParams) -> Result<()> {
        ctx.accounts.ONft_config.bump = ctx.bumps.ONft_config;
        ctx.accounts.ONft_config.token_mint = ctx.accounts.token_mint.key
            ();
        ctx.accounts.ONft_config.ext = ONftConfigExt::Adapter(ctx.
            accounts.token_escrow.key());
        ctx.accounts.ONft_config.token_program = ctx.accounts.
            token_program.key();

        ctx.accounts.lz_receive_types_accounts.ONft_config = ctx.accounts
            .ONft_config.key();
        ctx.accounts.lz_receive_types_accounts.token_mint = ctx.accounts.
            token_mint.key();

        let oapp_signer = ctx.accounts.ONft_config.key();
        ctx.accounts.ONft_config.init(
            params.endpoint_program,
            params.admin,
            params.shared_decimals,
            ctx.accounts.token_mint.decimals,
            ctx.remaining_accounts,
            oapp_signer,
        )
    }
}
```

Recommendation

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.

Use this function upon `onft_config` creation to prevent the mint close authority extension :

```rust
pub fn is_supported_mint(mint_account: &InterfaceAccount<Mint>) -> bool {
    let mint_info = mint_account.to_account_info();
    let mint_data = mint_info.data.borrow();
    let mint = StateWithExtensions::<spl_token_2022::state::Mint>::unpack
       (&mint_data).unwrap();

    let extensions = mint.get_extension_types().unwrap();
    for e in extensions {
        if e == ExtensionType::MintCloseAuthority {
            return false;
        }
    }

    true
}
```

Then revert if the mint extension is not supported:

```rust
if !is_supported_mint(&ctx.accounts.mint) {
    return Err(Error::UnsupportedBaseMint.into());
}
```

Team Response

Fixed.

[H-01] Denial of Service Risk Due to Frozen `token_escrow` in `ONFT::Adapter`

Severity

High Risk

Description

In the `init_ONft` 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 ONFT.

```
impl InitAdapterONft<_> {
  pub fn apply(ctx: &mut Context<InitAdapterONft>, params: &
     InitAdapterONftParams) -> Result<()> {
      ctx.accounts.ONft_config.bump = ctx.bumps.ONft_config;
      ctx.accounts.ONft_config.token_mint = ctx.accounts.token_mint.key()
         ;
      ctx.accounts.ONft_config.ext = ONftConfigExt::Adapter(ctx.accounts.
         token_escrow.key());
      ctx.accounts.ONft_config.token_program = ctx.accounts.token_program
         .key();

      ctx.accounts.lz_receive_types_accounts.ONft_config = ctx.accounts.
         ONft_config.key();
      ctx.accounts.lz_receive_types_accounts.token_mint = ctx.accounts.
         token_mint.key();
```

```
      let oapp_signer = ctx.accounts.ONft_config.key();
      ctx.accounts.ONft_config.init(
          params.endpoint_program,
          params.admin,
          params.shared_decimals,
          ctx.accounts.token_mint.decimals,
          ctx.remaining_accounts,
          oapp_signer,
      )
```

Impact

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

```
match &ctx.accounts.ONft_config.ext {
    ONftConfigExt::Adapter(_) => {
        if let Some(escrow_acc) = &mut ctx.accounts.token_escrow {
            // lock
            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: escrow_acc.to_account_info(),
                        authority: ctx.accounts.signer.to_account_info(),
                    },
                ),
                amount_sent_ld,
                ctx.accounts.token_mint.decimals,
            )?;
        } else {
            return Err(ONftError::InvalidTokenEscrow.into());
        }
    },
```

Recommendation

1. During ONFT 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.

Add this check in the `init_ONFT` function to the mint account to prevent using tokens with freeze authority:

```
if mint_account.freeze_authority.is_some() {
    return Err(Error::MintHasFreezeAuthority);
}
```

Team Response

Fixed.

[M-01] Unchecked Math Operations Leading to Potential Arithmetic Errors

Severity

Medium Risk

Description

There are multiple instances where unchecked math operations are used, which can result in arithmetic errors or overflow. Below is an example of unchecked math:

```
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;
}
```

In contrast, checked math is applied in other parts of the code, as shown here:

```
match self.tokens.checked_sub(amount) {
    Some(new_tokens) => {
        self.tokens = new_tokens;
        Ok(())
    },
    None => Err(error!(ONftError::RateLimitExceeded)),
}
```

Unchecked math can lead to potential overflow or underflow, resulting in unintended behaviour or errors within the protocol.

Recommendation

Ensure the use of checked math operations (e.g., `checked_sub`, `checked_mul`) to prevent overflow and arithmetic errors. This will safeguard the operations and minimize the risk of errors.

```
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;
+   new_tokens = new_tokens.checked_add(time_elapsed_in_seconds.
   checked_mul(self.refill_per_second)?).ok_or_else(|| ONftError::
   Overflow)?;
}
```

This approach handles potential overflows and returns an error if the arithmetic fails, ensuring safer operations.

Team Response

Fixed.

[M-02] Unlimited Decimals of Local Token Mints Result in DoS and Potential Overflow

Severity

Medium Risk

Description

In the `init()` function of the `onft_config`:

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

This code should cap the maximum number of 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 difference between the local and shared decimals, it can cause an overflow. The math used in `sd2ld` is not checked:

```
let amount_ld = ctx.accounts.ONft_config.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 as follows:

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

Impact

The impact can be:

1. **DoS**: The sending and receiving functions may fail if the token with large decimals cannot fit into the `u64` datatype.
2. **Overflow**: Since the `sd2ld` function does not use checked math, an overflow may occur.

Proof of Concept

If the local token has 18 decimals and the shared token has 6 decimals, the `ld2sd_rate` will be `10^(12)`. If an amount of `10_000 * 10^(6)` is received, the amount calculated from `sd2ld` will be `10_000 * 10^(18)`. This value cannot be stored in a `u64`, causing the protocol to fail when attempting to send or receive tokens with such large decimals.

Recommendation

Add a decimal limit in the `init_onft` instruction, allowing only tokens with appropriate decimals that can be stored in a `u64`.

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

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

Team Response

Fixed

[M-03] Missing Size Checks for `compose_msg` can Lead to Oversized Messages and Transaction Failures

Severity

Medium Risk

Description

The ONFT implementation currently lacks explicit size checks for the `compose_msg` parameter in its encoding functions. This omission could potentially lead to the creation of oversized messages, which might cause issues with cross-chain communication and Solana transaction processing. Solana imposes a maximum transaction size limit of 1232 bytes, and exceeding this size can result in failed transactions.

The vulnerable code resides primarily in the `msg_codec.rs` and `compose_msg_codec.rs` files:

Affected Code Snippets

Encoding Function (from `msg_codec.rs`):

```rust
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 {
        let mut encoded = Vec::with_capacity(40); // 32 + 8
        encoded.extend_from_slice(&send_to);
        encoded.extend_from_slice(&amount_sd.to_be_bytes());
        encoded
    }
}
```

Receiving Function (from `Lz_receive.rs`):

```rust
if let Some(message) = msg_codec::compose_msg(&params.message) {
    oapp::endpoint_cpi::send_compose(
        ctx.accounts.ONft_config.endpoint_program,
        ctx.accounts.ONft_config.key(),
        &ctx.remaining_accounts[Clear::MIN_ACCOUNTS_LEN..],
        seeds,
        SendComposeParams {
            to: to_address,
            guid: params.guid,
            index: 0, // only 1 compose msg per lzReceive
            message: compose_msg_codec::encode(
                params.nonce,
                params.src_eid,
                amount_received_ld,
                &message,
            ),
        },
    )?;
}
```

Without proper size limitations, excessively large `compose_ms` ' values can easily breach transaction size limits, potentially resulting in DoS (Denial of Service) vulnerabilities during cross-chain operations or failed transactions on Solana.

Impact

- **Cross-chain communication failures**: Oversized messages can prevent successful transaction execution, especially when the total transaction size exceeds Solana's 1232-byte limit.
- **DoS risk**: Repeated oversized messages could result in denial of service, as nodes might reject oversized transactions or consume excessive resources attempting to process them.

Recommendation

Implement a maximum size limit for the `compose_msg` parameter to avoid oversized messages. Add a constant to enforce the limit during message composition.

Define a constant for the maximum allowed size:

```
const MAX_MSG_SIZE: usize = 1024; // Adjust based on protocol
    requirements
```

Modify the `encode` function to include a size check:

```
pub fn encode(
    send_to: [u8; 32],
    amount_sd: u64,
    sender: Pubkey,
    compose_msg: &Option<Vec<u8>>,
) -> Result<Vec<u8>, OFTError> {
    if let Some(msg) = compose_msg {
        if msg.len() > MAX_MSG_SIZE {
            return Err(OFTError::MessageTooLarge);
        }
        let mut encoded = Vec::with_capacity(72 + msg.len());
        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);
        Ok(encoded)
    } else {
        let mut encoded = Vec::with_capacity(40);
        encoded.extend_from_slice(&send_to);
        encoded.extend_from_slice(&amount_sd.to_be_bytes());
        Ok(encoded)
    }
}
```

This approach ensures that message sizes remain within safe limits, preventing transaction failures or potential DoS attacks.

Team Response

Fixed.

[L-01] Require New Admin as Co-signer for Admin Setting in `transfer_admin` Instruction

Severity

Low Risk

Description

The admin can mistakenly be set to an invalid address because there is no validation or two-step admin transfer process. This could lead to losing control over the onft_config if the admin is set to an invalid address. To prevent this, the new admin should be made a co-signer of this call to ensure an intentional and valid transfer of admin rights.

Recommendation

Require the new admin to be a co-signer of this call to prevent accidental admin transfers and implement a two-step process. Separating the admin setting into a dedicated instruction with signature verification would enhance security.

```
pub struct TransferAdmin<info> {
    pub admin: Signer<info>,
    #[account(
        mut,
        seeds = [ONft_SEED, &get_ONft_config_seed(&ONft_config).to_bytes
           ()],
        bump = ONft_config.bump,
        has_one = admin @ONftError::Unauthorized
    )]
    pub ONft_config: Account<info, ONftConfig>,
++   pub new_admin: Signer<info>,
}
impl TransferAdmin<_> {
    pub fn apply(ctx: &mut Context<TransferAdmin>, params: &
       TransferAdminParams) -> Result<()> {
--        ctx.accounts.ONft_config.admin = params.admin;
++  ctx.accounts.ONft_config.admin = new_admin;
        Ok(())
    }
}
```

This would ensure that both the current and new admins are part of the transaction, reducing the risk of misconfiguration.

Team Response

Fixed.

[L-02] Inconsistent compose_msg Function Implementation May Lead to Silent Failures

Severity

Low Risk

Description

ONFT 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 compose 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.

Recommendations

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>>`:

Team Response

Fixed.

[L-03] Lack of Pause Functionality for Cross-Chain Token Transfers Could Lead to Security Risks

Severity

Low Risk

Description

Protocol administrators should have the ability to pause cross-chain token transfers if any malicious activity is detected. It is a common security practice to implement a pause flag in the `onft_config` structure and assign a `pauser` role to control this functionality. This feature is crucial for handling emergency situations and preventing potential exploits.

Recommendation

Introduce a `paused` flag in the `onft_config` to ensure that cross-chain functionality for a specific token can be paused when necessary. Additionally, assign a `pauser` role to control this mechanism.

```
pub struct ONftConfig {
    // immutable
    pub ld2sd_rate: u64,
    pub token_mint: Pubkey,
    pub token_program: Pubkey,
    pub endpoint_program: Pubkey,
    pub bump: u8,
    // mutable
    pub admin: Pubkey,
    pub ext: ONftConfigExt,
+   pub pauser: Pubkey,
+   pub paused: bool,
}
```

Add the following check at the beginning of the `send` and `lz_receive` functions to ensure the token is not paused during execution:

```
require!(!ctx.accounts.onft_config.paused, OFTError::Paused);
```

Team Response

Acknowledged.

shieldify

Thank you!