shieldify

Adrastea LRT

SECURITY REVIEW

Date: 26 October 2024

CONTENTS

1. **About Shieldify Security** 3
2. **Disclaimer** 3
3. **About Adra LRT** 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** 5

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 Adra LRT

Adrastea's main purpose is ultra-accessible real yields on Solana. As a restaking protocol, an input token and an output token (LRT) are needed for the restaking pool. The input token can be any form of asset the pool takes and the output token should be the liquid restaking token issued by the restaking pool. Users transfer the input tokens to the restaking pool and should get the output token (LRT) back. Under the hood, the restaking pool delegates authority delegates/undelegated the input token to AVSs and manages the AVS token. Users should be able to withdraw their funds anytime, even if there is no enough input token liquidity in the restaking pool.

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 6 days with a total of 48 hours dedicated by `0xcastle_chain` from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying five Low severity issues around arbitrary input tokens, token extensions leading to invariant manipulation, and denial of service due to the risk of input token mint being led by freeze authority.

5.1 Protocol Summary

Project Name	**Adra LRT**
Repository	adra-lrt
Type of Project	DeFi, Liquid restaking protocol
Audit Timeline	6 days
Review Commit Hash	47ce20e98b23ea7e5e3fb4d9eef089f201de2218
Fixes Review Commit Hash	N/A

5.2 Scope

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

File	LOC
programs/adra-lrt/src/utils.rs	6
programs/adra-lrt/src/lib.rs	70
programs/adra-lrt/src/errors.rs	28
programs/adra-lrt/src/state/lrt_pool.rs	12
programs/adra-lrt/src/state/mod.rs	2
programs/adra-lrt/src/contexts/delegate.rs	188
programs/adra-lrt/src/contexts/deposit.rs	110
programs/adra-lrt/src/contexts/initialize.rs	57
programs/adra-lrt/src/contexts/mod.rs	20

programs/adra-lrt/src/contexts/transfer_delegate_authority.rs	25
programs/adra-lrt/src/contexts/update_mint_limit.rs	23
programs/adra-lrt/src/contexts/withdraw.rs	109
programs/adra-lrt/src/contexts/withdraw_stake.rs	201
Total	**851**

6. Findings Summary

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

- **Low** issues: **5**
- **Info** issues: **1**

ID	Title	Severity	Status
[L-01]	Arbitrary Input Tokens and Token Extensions Leading to Invariant Manipulation	Low	Acknowledged
[L-02]	Risk of Input Token Mint with Freeze Authority Leading to Permanent DoS	Low	Acknowledged
[L-03]	Require New Authority as Co-Signer for Authority Transmission	Low	To be fixed
[L-04]	Insufficient Account Size Checks and Lack of Reallocation Support	Low	To be fixed
[L-05]	Missing Event Emissions for State-Changing Functions	Low	To be fixed
[I-01]	Add Pauser Role to Pool Struct for Emergency Pausing	Info	To be fixed

7. Findings

[L-01] Arbitrary Input Tokens and Token Extensions Leading to Invariant Manipulation

Severity

Low Risk

Description

The current implementation allows the input token mint to have a freeze authority, which can result in a potential denial of service (DoS) attack on the pool. If the input token mint's freeze authority exercises control, the pool's input token vault can be frozen, leading to a **permanent loss of funds** for users. This is a critical issue, as frozen token accounts cannot transfer tokens, rendering the pool inoperable.

Here's the relevant code:

```
#[account(
    mint::token_program = token_program,
)]
input_token_mint: Box<InterfaceAccount<'info, Mint>>,

#[account(
    mut,
    associated_token::authority = pool,
    associated_token::mint = input_token_mint,
    associated_token::token_program = token_program
)]
pool_input_token_vault: Box<InterfaceAccount<'info, TokenAccount>>,
```

The pool relies on the assumption that the input token mint is safe, but if the input token mint has a freeze authority, that authority can freeze the pool's input token vault. This could prevent further deposits, withdrawals, or transfers, effectively freezing user funds.

Impact

- **Denial of Service (DoS)**: The pool's input token vault can be frozen by the mint's freeze authority, halting all operations that involve this vault.
- **Permanent Loss of Funds**: If the vault is frozen, users will not be able to withdraw their funds, leading to a potential permanent loss of funds if the freeze is never lifted.

Proof of Concept

The following tests demonstrate how to simulate this attack:

```
#[cfg(test)]
mod tests {
    use solana_program::pubkey::Pubkey;
    use solana_program_test::*;
    use solana_sdk::{
        signature::{Keypair, Signer},
        transaction::Transaction,
        system_instruction,
    };
```

```rust
use spl_token_2022::{
    instruction::{initialize_mint2, close_account, set_authority,
        initialize_mint_close_authority},
    extension::ExtensionType,
    state::Mint,
};
use solana_program::program_pack::Pack; // Import the Pack trait for
    Mint::LEN

async fn create_mint(
    banks_client: &mut BanksClient,
    payer: &Keypair,
    mint_keypair: &Keypair,
    mint_authority: &Keypair,
    freeze_authority: Option<&Pubkey>,
    decimals: u8,
) -> Result<(), Box<dyn std::error::Error>> {
    // Calculate the space required using the ExtensionType
    let space = ExtensionType::try_calculate_account_len::<Mint>(&[
        ExtensionType::MintCloseAuthority]).unwrap();
    // Use Mint::LEN from the Pack trait
    let mint_account_rent = banks_client
        .get_rent()
        .await?
        .minimum_balance(space);

    let recent_blockhash = banks_client.get_latest_blockhash().await
        ?;

    // Step 1: Create the account for the mint with the required size
        and rent-exempt balance
    let create_account_tx = Transaction::new_signed_with_payer(
        &[system_instruction::create_account(
            &payer.pubkey(),
            &mint_keypair.pubkey(),
            mint_account_rent,
            space as u64,
            &spl_token_2022::id(),
        )],
        Some(&payer.pubkey()),
        &[payer, mint_keypair],
        recent_blockhash,
    );

    banks_client.process_transaction(create_account_tx).await?;
```

```
    // Step 2: Initialize the mint with initialize_mint2
    let recent_blockhash = banks_client.get_latest_blockhash().await
        ?;
    let initialize_mint_tx = Transaction::new_signed_with_payer(
        &[
            initialize_mint_close_authority(
                &spl_token_2022::id(),
                &mint_keypair.pubkey(),
                Some(&mint_authority.pubkey()),
            )?,
            initialize_mint2(
                &spl_token_2022::id(),
                &mint_keypair.pubkey(),
                &mint_authority.pubkey(),
                freeze_authority,
                decimals,
            )?
        ],
        Some(&payer.pubkey()),
        &[payer],
        recent_blockhash,
    );

    banks_client.process_transaction(initialize_mint_tx).await?;
    Ok(())
}

async fn close_mint_account(
    banks_client: &mut BanksClient,
    payer: &Keypair,
    mint_keypair: &Keypair,
    receiver: &Pubkey,
    mint_authority: &Keypair,
) -> Result<(), Box<dyn std::error::Error>> {
    let recent_blockhash = banks_client.get_latest_blockhash().await
        ?;
    let tx = Transaction::new_signed_with_payer(
        &[close_account(
            &spl_token_2022::id(),
            &mint_keypair.pubkey(),
            receiver,
            &mint_authority.pubkey(),
            &[&mint_authority.pubkey()],
        )?],
        Some(&payer.pubkey()),
        &[payer, mint_authority],
        recent_blockhash,
    );
    banks_client.process_transaction(tx).await?;
    Ok(())
}
```

```rust
#[tokio::test]
async fn test_initialize_mint_and_close() {
    // Register the spl_token_2022 program processor
    let program_test = ProgramTest::new(
        "spl_token_2022",
        spl_token_2022::id(),
        processor!(spl_token_2022::processor::Processor::process)
    );

    // Start the test environment
    let (mut banks_client, payer, _recent_blockhash) = program_test.
       start().await;

    // Create keypairs for the mint and the mint authority
    let mint_keypair = Keypair::new();
    let mint_authority = Keypair::new();
    let receiver_account = Keypair::new(); // Account that will
       receive the remaining rent

    // Step 1: Initialize the mint using spl_token_2022
    create_mint(
        &mut banks_client,
        &payer,
        &mint_keypair,
        &mint_authority, // Correctly use the public key of
           mint_authority
        Some(&mint_authority.pubkey()), // Freeze authority
        0, // Token decimals
    )
    .await
    .unwrap();

    // // Fetch and print the current decimals value
    // let mint_data = get_mint_data(&mut banks_client, &mint_keypair
       .pubkey()).await.unwrap();
    // println!("Old Mint Decimals: {}", mint_data.decimals);


    // Step 2: Close the mint account
    close_mint_account(
        &mut banks_client,
        &payer,
        &mint_keypair,
        &receiver_account.pubkey(), // Receiver of the rent remains
        &mint_authority, // Correct mint authority keypair is used
           here
    )
    .await
    .unwrap();
```

```
        // Step 3: re-initialize the mint account with different decimal

        create_mint(
            &mut banks_client,
            &payer,
            &mint_keypair,
            &mint_authority, // Correctly use the public key of
                mint_authority
            Some(&mint_authority.pubkey()), // Freeze authority
            6, // Token decimals
        )
        .await
        .unwrap();
    }
}
```

Recommendations

1. **Validate Absence of Freeze Authority**: Ensure that the input token mint does not have a freeze authority or that the pool controls the freeze authority, preventing any external actor from freezing the pool's input token vault.

   Example:

   ```
   #[account(
       mint::freeze_authority = COption::None, // Ensure no freeze
           authority exists
       mint::token_program = token_program,
   )]
   input_token_mint: Box<InterfaceAccount<'info, Mint>>,
   ```

2. **Display Warnings for Tokens with `freeze_authority`:** If supporting tokens with a `freeze_authority` is necessary, display a warning to users in the UI, informing them of the risks associated with trading or interacting with these tokens.
3. **Implement Allowlist for Trusted Tokens:** For regulated stablecoins like USDC, which have a `freeze_authority` for security reasons, implement an allowlist for trusted tokens while applying strict checks on other tokens. This ensures the protocol can support widely-used tokens while minimizing risk.

Team Response

For the official pools managed and supported by the Adrastea Team, all input mint tokens used in each pool need to be compatible with Solayer's endoAVS to be considered, then they will undergo a comprehensive review process to mitigate the risk of any malicious behavior. This process will involve examining the token program utilized, the enabled extensions, freeze/mint authorities, and the overall trustworthiness of the token. Only the official pools will be displayed on our website, and the addresses of these official pools along with the related token mints will be provided in our documentation. We strongly recommend that users interact exclusively with the official pools.

[L-02] Risk of Input Token Mint with Freeze Authority Leading to Permanent DoS

Severity

Low Risk

Description

The current implementation allows the input token mint to have a freeze authority, which can result in a potential denial of service (DoS) attack on the pool. If the input token mint's freeze authority exercises control, the pool's input token vault can be frozen, leading to a **permanent loss of funds** for users. This is a critical issue, as frozen token accounts cannot transfer tokens, rendering the pool inoperable.

Here's the relevant code:

```
#[account(
    mint::token_program = token_program,
)]
input_token_mint: Box<InterfaceAccount<'info, Mint>>,

#[account(
    mut,
    associated_token::authority = pool,
    associated_token::mint = input_token_mint,
    associated_token::token_program = token_program
)]
pool_input_token_vault: Box<InterfaceAccount<'info, TokenAccount>>,
```

The pool relies on the assumption that the input token mint is safe, but if the input token mint has a freeze authority, that authority can freeze the pool's input token vault. This could prevent further deposits, withdrawals, or transfers, effectively freezing user funds.

Impact

- **Denial of Service (DoS)**: The pool's input token vault can be frozen by the mint's freeze authority, halting all operations that involve this vault.
- **Permanent Loss of Funds**: If the vault is frozen, users will not be able to withdraw their funds, leading to a potential permanent loss of funds if the freeze is never lifted.

Recommendations

1. **Validate Absence of Freeze Authority**: Ensure that the input token mint does not have a freeze authority or that the pool controls the freeze authority, preventing any external actor from freezing the pool's input token vault.

   Example:

   ```
   #[account(
       mint::freeze_authority = COption::None, // Ensure no freeze
          authority exists
       mint::token_program = token_program,
   )]
   input_token_mint: Box<InterfaceAccount<'info, Mint>>,
   ```

Team Response

See response for [L-01].

[L-03] Require New Authority as Co-Signer for Authority Transmission

Severity

Low Risk

Description

The current `TransferDelegateAuthority` function allows for the transfer of delegate authority without any validation or a two-step confirmation process. This lack of validation can lead to critical errors, such as mistakenly transferring the delegate authority to an invalid or unintended address. As a result, control over the pool could be lost, leading to severe consequences for the protocol.

In the current implementation:

```
pub struct TransferDelegateAuthority<'info> {
    #[account(mut)]
    authority: Signer<'info>,

    #[account(
        mut,
        seeds = [b'lrt_pool', pool.output_token_mint.key().as_ref()],
        bump = pool.bump,
        constraint = pool.delegate_authority == authority.key()
    )]
    pool: Account<'info, LRTPool>,
    new_authority: UncheckedAccount<'info>,
}

impl<'info> TransferDelegateAuthority<'info> {
    pub fn transfer_authority(&mut self) -> Result<()> {
        self.pool.delegate_authority = self.new_authority.key();
        Ok(())
    }
}
```

There is no validation that the `new_authority` is intentional or valid, nor is there a requirement for the new authority to approve the transfer.

Impact

If the delegate authority is mistakenly set to an invalid address, the protocol would lose control over the pool's functions This could lead to :

- Permanent loss of control over the pool and prevent delegations

Recommendation

To prevent accidental transfers and loss of control, introduce the following improvements:

1. **Require the New Authority as a Co-Signer**: To ensure that the transfer is intentional, the new authority must also sign the transaction, proving they are aware of and agree to take over the role.
2. **Implement a Two-Step Transfer Process**: Split the admin transfer into two steps: first, initiating the transfer and, second, confirming the transfer by the new authority.

Team Response

We agree that implementing this feature would significantly improve the operation and management of the contracts. We will consider its inclusion in a future iteration.

[L-04] Insufficient Account Size Checks and Lack of Reallocation Support

Severity

Low Risk

Description

The protocol currently relies on hardcoded account sizes, which limits flexibility when updating or upgrading the account structures in the future. The lack of proper account size checks or reallocation support could result in accounts being too small to accommodate new fields or upgrades. For example, if the structure of `LRTPool` is updated, but the account size is fixed, it would lead to failures in adding new data without reallocation.

The protocol needs to account for Solana's support for account data reallocation to ensure that future updates are not restricted by insufficient space.

Impact

- **Upgrade Limitations**: Without flexible account sizing, future upgrades that require adding fields to structures like `LRTPool` could fail due to insufficient space in accounts.
- **Account Initialization Failures**: When account sizes are not checked or adjusted dynamically, any changes in the structure would necessitate manual resizing and redeployment, causing operational inefficiencies.

Recommendation

- **Add Padding Space**: Introduce extra padding in account structures like `LRTPool` to accommodate future upgrades without requiring immediate reallocation.

```rust
pub struct LRTPool {
    pub bump: u8,
    pub input_token_mint: Pubkey,
    pub output_token_mint: Pubkey,
    pub delegate_authority: Pubkey,
    pub output_mint_limit: u64,
    pub padding: [u8; 64],  // Padding to support future upgrades
}
```

- **Increase Initial Space Allocation**: Adjust the space allocated during account initialization to account for this padding, ensuring there's enough buffer for future changes. This can be done by increasing the `init` space parameter during account creation.
- **Dynamic Size Checks and Reallocation**: Implement checks for account size limits and enable reallocation to expand account sizes dynamically as needed for updates or protocol changes.

Team Response

See response for [L-03].

[L-05] Missing Event Emissions for State-Changing Functions

Severity

Low Risk

Description

The protocol's state-changing functions, such as `stake`, `deposit`, and `withdraw`, do not emit events after execution. Events are critical for notifying off-chain components (such as explorers, indexers, and dApps) of changes to account states. Without events, it becomes challenging for external systems to track state changes, which can lead to inefficiencies in querying or monitoring the protocol.

For example, in the `stake` and `deposit` functions, there are no event emissions, making it difficult to track when tokens are staked, deposited, or withdrawn from the pool.

Impact

- **Lack of Transparency**: Off-chain components will not be able to listen to state changes efficiently, making it harder to track or monitor critical actions such as token deposits or staking.
- **Operational Inefficiencies**: Off-chain services may need to repeatedly query on-chain data, increasing the load on the network and slowing down interactions.

Recommendation

1. **Emit Events for State Changes**: Ensure that every function that changes the state (e.g., `stake`, `deposit`, `withdraw`, `delegate`) emits an event after completing the action. This event should include relevant data such as the amount staked or withdrawn, the accounts involved, and the resulting new state.

Example for `deposit` function:

```
#[event]
pub struct DepositEvent {
    pub depositor: Pubkey,
    pub amount: u64,
    pub pool: Pubkey,
    pub timestamp: i64,
}

pub fn deposit(ctx: Context<Deposit>, amount: u64) -> Result<()> {
    // transfer input token into the pool
    ctx.accounts.stake(amount)?;

    // calculate mint amount
    let mint_amount = ctx.accounts.calculate_output_token_amount(amount);

    // mint output token
    ctx.accounts.mint_output_token(mint_amount)?;

    // emit the event
    emit!(DepositEvent {
        depositor: ctx.accounts.signer.key(),
        amount,
        pool: ctx.accounts.pool.key(),
        timestamp: Clock::get()?.unix_timestamp,
    });

    Ok(())
}
```

2. **Add Similar Events for Other Functions**: Implement similar events for other state-changing functions like `withdraw` and `delegate` to ensure all state changes are observable by off-chain components.

Team Response

See response for [L-03].

[I-01] Add Pauser Role to Pool Struct for Emergency Pausing

Severity

Info

Description

The current pool struct does not include any mechanism for pausing operations in case of malicious activity. By adding a `pauser` role and a `paused` flag, it will be possible to temporarily suspend key functions such as deposits, withdrawals, and staking. This enhancement allows administrators to quickly react to malicious actions or protocol vulnerabilities and halt operations until the issue is resolved.

Here's the original struct:

```
pub struct LRTPool {
    pub bump: u8,
    pub input_token_mint: Pubkey,
    pub output_token_mint: Pubkey,
    pub delegate_authority: Pubkey, // The one who handles delegation
    pub output_mint_limit: u64,
}
```

Recommendation

1. **Add a Pauser Role**: Introduce a new `pauser` role in the `LRTPool` struct that will be authorized to pause and resume operations.

   ```
   pub pauser: Pubkey, // The address that can pause the pool
   ```

2. **Add a Paused Flag**: Include a `paused` boolean flag to check if the pool is paused before allowing deposits, withdrawals, or staking.

   ```
   pub paused: bool, // Flag indicating whether deposits/withdrawals are
       paused
   ```

3. **Check the Paused Flag in Functions**: Before allowing deposits, withdrawals, or staking, ensure the logic checks whether the `paused` flag is set to `true`. If it is, halt the operation.

   Example in a deposit function:

   ```
   if pool.paused {
       return Err(LRTPoolError::PoolPaused); // Custom error for paused
           state
   }
   ```

Updated Struct Example:

```
pub struct LRTPool {
    pub bump: u8,
    pub input_token_mint: Pubkey,
    pub output_token_mint: Pubkey,
    pub delegate_authority: Pubkey,
    pub output_mint_limit: u64,
    pub pauser: Pubkey,       // The one who can pause the pool
    pub paused: bool,         // Flag to indicate if operations are paused
}
```

This addition provides a safeguard to prevent malicious actions from causing further harm, as the pool's operations can be quickly paused in the event of suspicious or dangerous activity.

Team Response

See response for [L-03].

shieldify

Thank you!