Acala
Findings & Analysis Report

2024-05-02

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Acala smart contract system written in Rust. The audit took place between March 22 — April 5, 2024.

Wardens

19 Wardens contributed reports to Acala:

1. carrotsmuggler
2. ZanyBonzy
3. zhaojie
4. AM (StefanAndrei and 0xmatei)
5. ihtishamsudo
6. TheSchnilch
7. Bauchibred
8. djxploit
9. Aymen0909
10. n4nika

11. DarkTower (0xrex and haxatron)
12. ABAIKUNANBAEV
13. 0xTheC0der
14. hunter_w3b
15. 0xepley
16. LinKenji
17. Cryptor

This audit was judged by Lambda.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 7 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 7 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Acala repository, and is composed of 3 smart contracts written in the Rust programming language and includes 1135 lines of Rust code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (3)

[H-01] `transfer_share_and_rewards` allows for self transfer

Submitted by ZanyBonzy, also found by ihtishamsudo

The rewards library holds the `transfer_share_and_rewards` allows for self transfer which can be used to double shares and rewards. Important to note that the function, for now is not in use by the in-scope contracts. However, I still believe it's worth pointing out.

Proof of Concept

Copy and paste the below test into tests.rs. It shows how users Alice and Bob, by invoking this function, can increase their share/rewards balance.

```
	fn transfer_to_self() {
		ExtBuilder::default().build().execute_with(|| {
			// Open a pool with bob holding 100 share
			RewardsModule::add_share(&BOB, &DOT_POOL,
			// Accumulate rewards
			assert_ok!(RewardsModule::accumulate_rewa
			// Alice deposits into the pool and gets
			RewardsModule::add_share(&ALICE, &DOT_POO
			// Assert that rewards still exist
			assert_ok!(RewardsModule::accumulate_rewa
			// Gets pools info
			let pool_info = RewardsModule::pool_infos
			// Ensures that reward transfer doesn't a
			let new_pool_info = RewardsModule::pool_i
			assert_eq!(pool_info, new_pool_info, "rew
			// Assert that Alice's share/rewards tran
			assert_ok!(RewardsModule::transfer_share_
			// Assert that alice's share/reward balan
			assert_eq!(
				RewardsModule::shares_and_withdra
				(190, vec![(NATIVE_COIN, 190)].in
			);
			// Alice has discovered infinite money gl
			// Alice's transfers shares and rewards a
			assert_ok!(RewardsModule::transfer_share_
			// Assert that her share/reward balance s
			assert_eq!(
				RewardsModule::shares_and_withdra
				(370, vec![(NATIVE_COIN, 370)].in
			);
			// She transfers a some of her shares/rew
			assert_ok!(RewardsModule::transfer_share_
			assert_eq!(
				RewardsModule::shares_and_withdra
				(300, vec![(NATIVE_COIN, 300)].in
			);
			assert_eq!(
				RewardsModule::shares_and_withdra
				(170, vec![(NATIVE_COIN, 70)].int
                    );
                    // Bob decides to try it out himself, Ass
                    assert_ok!(RewardsModule::transfer_share_
                    assert_eq!(
                            RewardsModule::shares_and_withdra
                            (270, vec![(NATIVE_COIN, 111)].ir
                    );
                    assert_ok!(RewardsModule::transfer_share_
                    assert_eq!(
                            RewardsModule::shares_and_withdra
                            (370, vec![(NATIVE_COIN, 152)].ir
                    );
            });
    }
```

Recommended Mitigation Steps

Include a check in the function that returns if `who == other`.

[Lambda (judge) increased severity to High]()

[xlc (Acala) confirmed and commented]():

> Fixed by [this PR](). Just want to highlight that `transfer_share_and_rewards` is not currently used.

[H-02] Early user can break pool via inflation attack due to no minimum liquidity check in the incentive contract

Submitted by [carrotsmuggler](), *also found by* [zhaojie]()

The incentive contract does not enforce a minimum liquidity limit. This means users can have as little as 1 share in the pool. This can lead to inflation attacks as described below.

Let's imagine the state of the pool is as follows:

There is a single depositor, with 1000 shares deposited. Rewards have been accumulated up to 500 tokens. The user can then withdraw 998 shares, leaving 2 shares. They will also claim the rewards, and leave 1 reward tokens in the pool. This is the setup for the inflation attack. The user can then deposit 1 share.

The inflation is calculated as shown below:

```
U256::from(add_amount.to_owned().saturated_into::<u128>()
    .saturating_mul(total_reward.to_owned().saturated_in
    .checked_div(initial_total_shares.to_owned().saturate
    .unwrap_or_default()
    .as_u128()
    .saturated_into()
```

Here `total_reward=1`, `add_amount=1` and `initial_total_shares=2`. So the result is calculated to `0`; so inflation is `0`.

After this step, the `initial_total_shares` is updated to 3. Now the user can deposit 2 wei of shares without changing the inflation amount. Next iteration, they can deposit 4 shares. This way, the user can deposit `2**n` shares each iteration, and inflate the `initial_total_shares` without affecting the reward inflation. This leads to the situation where the `total_shares` keeps growing according to the deposit, but the entire reward inflation mechanism is broken. This lets users steal reward tokens from other users, and is a high severity issue.

In fact, whenever the `total_reward` value is less than the `total_shares`, this issue can be triggered. This is because in those conditions, users can create deposits and have the `reward_inflation` evaluate to `0`. `0` `reward_inflation` basically means later users can steal rewards of earlier users, as is outlined in the docs. However, this donation attack is more effective the lower the `total_shares` in the system.

Proof of Concept

The situation can be created via the following steps:

1. Since there is no minimum deposit, we can create a situation where the `total_reward < total_shares`, and `total_shares=2`. This also works for higher values of `total_share`, but is most potent in this stage.
2. User deposits 1 share, or any number of shares as long as `deposit*total_reward/total_shares` is less than 1. `reward_inflation` will be `0`, while the user gets their shares accounted for.
3. Since `total_shares` has now increased, the user can deposit more shares now, and still have the `reward_inflation` be `0`. This way, the user can keep depositing shares and increasing the `total_shares` without affecting the reward inflation.
4. Since `reward_inflation` and thus `total_reward` has not increased, but the `total_shares` have increased, users will lose rewards, since the rewards are calculated as `total_reward * user_shares / total_shares`. This means older users lose shares.

While this vector is generally applicable and can lead to small losses when there's a lot of liquidity, this becomes more potent when there is very low liquidity in the pool. This was the method of attack for the Wise Lending hack and is a high severity issue. More details can be found in the blog post here which outlines the attack scenario with more numbers and examples.

Tools Used

Substrate

Recommended Mitigation Steps

Add a minimum liquidity limit. This will ensure the pool never reaches a liquidity amount so low that rounding errors become significant.

xlc (Acala) confirmed and commented:

> It is actually almost impossible to trigger this in production because anyone can deposit into the incentives pool at any time. I.E. before rewards starts accumulates.

> Fixed by this PR.

[H-03] `transfer_share_and_rewards` can be used to transfer out shares without transferring reward debt due to rounding

Submitted by carrotsmuggler, also found by AM

The function `transfer_share_and_rewards` can be used to split up the position in a single account into multiple accounts. The contract sends some of the shares to be held by the second account, and similarly also updates the reward debt of the receiving account so that the receiver cannot take out more rewards than they deserve.

This is calculated in the following snippet. `move_balance` is the amount of the reward debt that is to be transferred to the receiver.

```
    let move_balance = U256::from(balance.to_owned().saturate
        * U256::from(move_share.to_owned().saturated_into::<u
        / U256::from(share.to_owned().saturated_into::<u128>(
```

Here we see the calculation is simple and by default is rounded down. So if `balance*move_share` is lower than `share`, `move_balance` evaluates to 0. So the receiving account's reward debt is not increased at all!

```
    increased_rewards
        .entry(*reward_currency)
        .and_modify(|increased_reward| {
            *increased_reward = increased_reward.saturating_a
```

Since `move_balance` is `0`, the `increased_reward` is not updated. This means the new account now has shares, but no reward debt. So the receiving account can claim rewards that were already claimed.

This can be done multiple times to drain the reward pool.

The criteria is that `balance*move_share` has to be lower than `share`. This can be achieved by sending a small fraction of the funds to the receiving account, such that `move_share` is much lower than `share`. Also, if `balance`, the reward debt of the sender is low, this facilitates the attack more.

Proof of Concept

A short POC is shown here demonstrating the issue. The attacker sends to the receiver a small share of their total. The receiver is shown to have no reward debt, while the sender does have reward debt. This shows that the receiver can claim rewards already claimed by the sender.

```
#[test]
fn test_rounding() {
ExtBuilder::default().build().execute_with(|| {
    RewardsModule::add_share(&ALICE, &DOT_POOL, 1000);
    assert_ok!(RewardsModule::accumulate_reward(&DOT_POOL
    // RewardsModule::add_share(&BOB, &DOT_POOL, 100);
    RewardsModule::claim_rewards(&ALICE, &DOT_POOL);

    let user_stat = RewardsModule::shares_and_withdrawn_r
    println!("ALICE stat before transfer: {:?}", user_sta
    let user_stat = RewardsModule::shares_and_withdrawn_r
    println!("BOB stat before transfer: {:?}", user_stat)

    assert_ok!(RewardsModule::transfer_share_and_rewards(

    let user_stat = RewardsModule::shares_and_withdrawn_r
    println!("ALICE stat after transfer: {:?}", user_stat
    let user_stat = RewardsModule::shares_and_withdrawn_r
    println!("BOB stat after transfer: {:?}", user_stat);
});
}
```

Output:

```
# Output is of the form ( ${share_balance}, {0: ${reward_
ALICE stat before transfer: (1000, {0: 100})
BOB stat before transfer: (0, {})
ALICE stat after transfer: (995, {0: 100})
BOB stat after transfer: (5, {0: 0})
test tests::test_rounding ... ok
```

The output shows that ALICE has 1000 shares and 100 reward debt, since ALICE just claimed her rewards. Alice sends BOB 5 shares. BOB ends up with 5 shares and 0 reward debt. So BOB can claim rewards again, even though it's the same money!

Tools Used

Substrate

Recommended Mitigation Steps

The calculation of `move_balance` should be changed to saturated round up instead of rounding down. This will ensure that the receiving account's reward debt is updated correctly. The `saturated` rounding up is important since the reward debt should never be larger than the reward pool, or it will cause underflow errors when subtracting.

Another option is to revert `transfer_share_and_rewards` operations if the reward debt of the receiving account is calculated to be `0`, unless the sending account ALSO has a reward debt of `0`.

Assessed type

Math

xlc (Acala) confirmed and commented:

> Just want to highlight that `transfer_share_and_rewards` is not currently used.

> We will choose to not fix this issue as the impact are relatively small and a complete fix is non-trivial. I don't think it is possible to make profit that is more than transaction fee anyway.

Medium Risk Findings (4)

[M-01] Claiming rewards while the deduction rate is `!= 0`, allows for repeated withdrawal of redistributed rewards

Submitted by n4nika, also found by ABAIKUNANBAEV and djxploit

If a participant in a pool claims rewards while the deduction rate is `!= 0`, the deducted rewards are redistributed between all the participants.

```
    fn payout_reward_and_reaccumulate_reward(
            pool_id: PoolId,
            who: &T::AccountId,
            reward_currency_id: CurrencyId,
            payout_amount: Balance,
            reaccumulate_amount: Balance,
    ) -> DispatchResult {
            if !reaccumulate_amount.is_zero() {
                    <orml_rewards::Pallet<T>>::accumulate_rew
            }
            T::Currency::transfer(reward_currency_id, &Self::
            Ok(())
    }
```

Since the deduction is redistributed between all participants (since `accumulate_reward` distributes to all participating users) including the one claiming the reward, they can repeatedly claim rewards and receive more of the rewards pool than they probably should.

Proof of Concept

```
#[test]
fn repeated_claiming() {
        ExtBuilder::default().build().execute_with(|| {
                assert_ok!(TokensModule::deposit(BTC, &V

                assert_ok!(IncentivesModule::update_clai
                        RuntimeOrigin::signed(ROOT::get(
                        vec![(PoolId::Dex(BTC_AUSD_LP), 
                ));

                assert_ok!(IncentivesModule::update_clai
                        RuntimeOrigin::signed(ROOT::get(
                        vec![(PoolId::Dex(BTC_AUSD_LP), 
                ));

                // 40% deduction rate
                assert_ok!(IncentivesModule::update_clai
                        RuntimeOrigin::signed(ROOT::get(
                        vec![(PoolId::Dex(BTC_AUSD_LP), 
                ));

                // give RewardsSource tokens
                TokensModule::deposit(BTC, &RewardsSourc

                // add 11 participants with equal shares
                TokensModule::deposit(BTC_AUSD_LP, &ALIC
                TokensModule::deposit(BTC_AUSD_LP, &BOB:


                // 1000 BTC as reward
                IncentivesModule::update_incentive_rewar

                // all 11 participants deposit their sha
                assert_ok!(IncentivesModule::deposit_dex
                assert_ok!(IncentivesModule::deposit_dex

                // first accumulation of rewards
                IncentivesModule::on_initialize(10);
                // ALICE claims their reward and should r
                assert_ok!(IncentivesModule::claim_rewarc
                let btc_before = TokensModule::total_bala

                // here ALICE should have claimed all the
                assert_ok!(IncentivesModule::claim_rewarc
                let btc_after = TokensModule::total_balar

                assert_eq!(btc_before, btc_after);
        });
    }
```

Recommended Mitigation Steps

Add the claiming user to `accumulate_rewards` and implement reaccumulating excluding the calling user.

xlc (Acala) disputed and commented:

> This is intended behaviour and non issue.

[M-02] Incentive accumulation can be sandwiched with additional shares to gain advantage over long-term depositors

Submitted by 0xTheC0der, also found by zhaojie, djxploit, and carrotsmuggler

Incentives are accumulated periodically in intervals of T::AccumulatePeriod blocks. Thereby, a **fixed** incentive reward amount, which is set via `update_incentive_rewards` (...), is accumulated among **all** deposited shares of a respective pool. This means if only `1` share is deposited, it is entitled for the all the rewards of this period, if `N` shares are deposited, the rewards are split among them, etc. (see PoC).

Furthermore, to be eligible for rewards, it is sufficient to deposit (DEX) shares before `accumulate_incentives` (...) is called via the `on_initialize` hook. This is also demonstrated in the `transfer_reward_and_update_rewards_storage_atomically_when_accumulate_incentives_work()` test case.

Afterward, rewards can be immediately claimed and (DEX) shares can be withdrawn **without** any unbonding period or other block/time related restrictions.

This leads to the following consequences:

- Users are **not** incentivized to keep (DEX) shares deposited, but rather to **sandwich** the incentive accumulation via deposit, claim & withdraw.
- Since (DEX) shares are only required for a short period of time (> 1 block) to perform the above action, an adversary can borrow vast amounts of (DEX) shares (or the underlying assets to get them) to crowd out honest users during the incentive accumulation and therefore obtain an unfairly high share of the fixed rewards.

For reference, see also the *earnings* module, where an unbonding period is explicitly enforced.

Proof of Concept

The *diff* below modifies the existing test case `transfer_reward_and_update_rewards_storage_atomically_when_accumulate_incentives_work()` to demonstrate that the *same* total reward amount is accumulated even if *twice* as many total shares are deposited. Therefore, it's possible to crowd out honest users with great short-term share deposits.

```
diff --git a/src/modules/incentives/src/tests.rs b/src/mo
index 1370d5b..fa16a08 100644
--- a/src/modules/incentives/src/tests.rs
+++ b/src/modules/incentives/src/tests.rs
@@ -1171,10 +1171,11 @@ fn transfer_reward_and_update_rew
                assert_eq!(TokensModule::free_balance(AUS

                RewardsModule::add_share(&ALICE::get(), &
+               RewardsModule::add_share(&BOB::get(), &Po
                assert_eq!(
                        RewardsModule::pool_infos(PoolId:
                        PoolInfo {
-                               total_shares: 1,
+                               total_shares: 2,
                                ..Default::default()
                        }
                );
@@ -1188,7 +1189,7 @@ fn transfer_reward_and_update_rewa
                assert_eq!(
                        RewardsModule::pool_infos(PoolId:
                        PoolInfo {
-                               total_shares: 1,
+                               total_shares: 2,
                                rewards: vec![(ACA, (30,
                        }
                );
@@ -1202,7 +1203,7 @@ fn transfer_reward_and_update_rewa
                assert_eq!(
                        RewardsModule::pool_infos(PoolId:
                        PoolInfo {
-                               total_shares: 1,
+                               total_shares: 2,
                                rewards: vec![(ACA, (60,
                        }
                );
```

Recommended Mitigation Steps

The present issue scales with the size of T::AccumulatePeriod in terms of blocks. Therefore, it's recommended (for fairness) to also track deposited shares in between the accumulation intervals and scale the incentive rewards according to the actual deposit duration.

Assessed type

MEV

xlc (Acala) disputed and commented:

> Furthermore, to be eligible for rewards, it is sufficient to deposit (DEX) shares before accumulate_incentives(...) is called via the on_initialize hook

> It is not possible to do perform user triggered action before `on_initialize`. To exploit this, attacker will need to acquire a large number of dex share somehow, deposit it, what for a block, withdraw it, ????, and repeat the same thing on next minute.

> Firstly, it is not possible to borrow such amount of share without some payments, because the lender have the full incentives to deposit the shares and getting the rewards. It is also not a lost free action to mint such amount of dex share and redeem them due to potential price exposure and sandwich risk.

> Therefore, it is economically impossible to exploit this behavior and make a profit.

Lambda (judge) commented:

> The described scenario has some stated assumptions with external requirements (availability of liquidity, possibility to front- and back-run the

> transactions risk free, etc...) that may not always hold in practice. But these are not very unreasonable assumptions and under these assumptions, a value leak is possible. This, therefore, fulfills the criteria of a valid medium according to the severity categorization.

[M-03] `Unbond_instant` removes incorrect amount of shares

Submitted by TheSchnilch*, also found by* Aymen0909

With `unbond_instant`, a user can unbond a bonded amount directly without having to wait. However, they must pay a fee for this:

```
    let amount = change.change;
    let fee = fee_ratio.mul_ceil(amount);
    let final_amount = amount.saturating_sub(fee);

    let unbalance = T::Currency::withdraw(&who, fee, Withdrav
    T::OnUnstakeFee::on_unbalanced(unbalance);

    T::OnUnbonded::happened(&(who.clone(), final_amount));
```

Here, the mistake is that `T::OnUnbonded::happened` is called with `final_amount`, meaning without a fee. As a result, a portion of the shares that were added as shares during bonding can no longer be removed. This, in turn, leads to these shares still receiving rewards that other users cannot receive.

If you insert a `println!` statement into the functions `bond` and `unbond_instant` to display the amounts with which `OnBonded::happened` and `OnUnbonded::happened` are called, you will see when you execute the following code that not all shares are removed when the bonded amount of a user is removed with `unbond_instant`:

For bond:

```
+ println!("change.change: {:?}", change.change);
144: T::OnBonded::happened(&(who.clone(), change.change)
145: Self::deposit_event(Event::Bonded {
146:    who,
147:    amount: change.change,
148: });
```

For `unbond_instant`:

```
+ println!("final_amount: {:?}", final_amount);
196: T::OnUnbonded::happened(&(who.clone(), final_amount
197: Self::deposit_event(Event::InstantUnbonded {
198:    who,
199:    amount: final_amount,
200:    fee,
201: });
```

Code that can be inserted into the file modules/earning/src/tests.rs to test both functions:

```
    #[test]
    fn poc() {
            ExtBuilder::default().build().execute_with(|| {
                    assert_ok!(Earning::bond(RuntimeOrigin::s

                    assert_ok!(Earning::unbond_instant(Runtim
            });
    }
```

The code can be started with the command `cargo test poc -- --nocapture`.

So every time a user calls `unbond_instant`, some shares will remain in the reward system and continue to accumulate rewards that other users who actually bond can no longer get. These shares can then no longer be removed and are stuck. With a maximum fee of 20%, some shares could accumulate over a period of time and these shares would accumulate rewards. The owners of the shares can still claim these rewards. In addition, one share is no longer the same as one underlying token.

Recommendation

For `unbond_instant`, the `final_amount` should not be used to remove the shares, but `change.change` should be used instead.

xlc (Acala) confirmed and commented:

> See the fix here.

[M-04] Storage can be bloated with low liquidity positions

Submitted by [ZanyBonzy](), *also found by* [Bauchibred]() *and* [carrotsmuggler]()

The `deposit_dex_share` function enforce no minimum amount that can be deposited into the pool allows for creating multiple pool positions. This causes that in a coordinated effort, for a pretty cheap cost, users/attackers can create multiple low liquidity positions to bloat the runtime storage. This is very important as substrate framework requires optimization of storage to prevent bloat which can lead to high maintenance costs for the chain and a potential DOS. A more in detail explanation can be found [here]().

Proof of Concept

The test case below shows how a user can create multiple 1 wei positions, and it can be added to [test.rs]().

```
#[test]
fn open_low_liquidity_positions() {
        ExtBuilder::default().build().execute_with(|| {
                assert_ok!(TokensModule::deposit(BTC_AUSD
                assert_eq!(TokensModule::free_balance(BTC
                assert_eq!(
                        TokensModule::free_balance(BTC_AU
                        0
                );
                assert_eq!(RewardsModule::pool_infos(Pool
                assert_eq!(
                        RewardsModule::shares_and_withdra
                        Default::default(),
                );
                assert_ok!(IncentivesModule::deposit_dex_
                        RuntimeOrigin::signed(ALICE::get(
                        BTC_AUSD_LP,
                        1
                ));
                assert_ok!(IncentivesModule::deposit_dex_
                        RuntimeOrigin::signed(ALICE::get(
                        BTC_AUSD_LP,
                        1
                ));
                assert_ok!(IncentivesModule::deposit_dex_
                        RuntimeOrigin::signed(ALICE::get(
                        BTC_AUSD_LP,
                        1
                ));
                assert_eq!(TokensModule::free_balance(BTC
                assert_eq!(
                        TokensModule::free_balance(BTC_AU
                        3
                );
                assert_eq!(
                        RewardsModule::pool_infos(PoolId:
                        PoolInfo {
                                total_shares: 3,
                                ..Default::default()
                        }
                );
                assert_eq!(
                        RewardsModule::shares_and_withdra
                        (3, Default::default())
                );
        });
}
```

Recommended Mitigation Steps

Introduce a minimum deposit amount.

xlc (Acala) confirmed and commented:

> Fixed here.

Low Risk and Non-Critical Issues

For this audit, 7 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Bauchibred** received the top score from the judge.

The following wardens also submitted reports: ZanyBonzy, n4nika, AM, 0xTheC0der, zhaojie, and Cryptor.

[01] Admin is a single point of failure

Multiple instances in code where the admin logic could lead to a brick in normal functionality.

For example take a look here.

```
/// The origin which may update incentive
type UpdateOrigin: EnsureOrigin<Self::Rur
```

This is an instance where a somewhat `admin` logic has been applied, note that this is passed in the config's constant paller. Now using this search command, we can see that there are three function calls where the expected caller is only accepted to be the `update incentive related params`.

Impact

Inaccess to update incentive related params if anything was to happen to the `UpdateOrigin`.

Recommended Mitigation Steps

Consider implementing a backdoor that the admins could use to change the incentive related params if anything were to happen to `UpdateOrigin` .

[02] Consider adding better documentation

This is very rampant in code, for example, take a look here.

```
pub fn add_share(who: &T::AccountId, pool: &T::P
        if add_amount.is_zero() {
                return;
        }

        PoolInfos::<T>::mutate(pool, |pool_info|
                let initial_total_shares = pool_
                pool_info.total_shares = pool_in

                let mut withdrawn_inflation = Ve

                pool_info
                        .rewards
                        .iter_mut()
                        .for_each(|(reward_curre
                                let reward_infla
                                        Zero::ze
                                } else {
                                        U256::fr
                                                .
                                                .
                                                .
                                                .
                                                .
                                };
                                *total_reward = 
                                *total_withdrawn_

                                withdrawn_inflat
                        });
                SharesAndWithdrawnRewards::<T>::m
                        *share = share.saturating
                        // update withdrawn infla
                        withdrawn_inflation
                                .into_iter()
                                .for_each(|(rewa
                                        withdrawn
                                                .
                                                .

                                                ]
                                                .
                                });
                });
        });
}
```

Evidently, this function is somewhat complex, i.e. the type of mathematical operations attached with it. However, that isn't the main case here. There are completely no documentations as to why some steps are taken, this heavily stalls the auditing process as we and other security researchers don't know what the intended behaviour is and can't really sit to break it.

Impact

Hard time understanding code for users/devs/auditors lead to bad integration.

Recommended Mitigation Steps

Consider adding better documentations to even if not all functions then at least the core/complex ones.

[03] Setters should always have equality checkers

Take a look here.

```
	pub fn set_share(who: &T::AccountId, pool: &T::Po
		let (share, _) = Self::shares_and_withdra

		if new_share > share {
			Self::add_share(who, pool, new_sh
		} else {
			Self::remove_share(who, pool, sha
		}
	}
```

This function sets a new share value for a pool, now there are no checks that `new_share != share` and as such whenever `new_share == share` the protocol unnecessarily (wrongly attempts to remove via `share.saturating_sub(new_share`).

Another instance can be seen here, where protocol in all instances of updating the amounts don't check if the provided amount is not already the stored amount and as such unnecessarily updates `v`, see here.

```
		pub fn update_incentive_rewards(
			origin: OriginFor<T>,
			updates: Vec<(PoolId, Vec<(Curren
		) -> DispatchResult {
			T::UpdateOrigin::ensure_origin(or
			for (pool_id, update_list) in upd
				if let PoolId::Dex(curren
					ensure!(currency_
				}

				for (currency_id, amount)
					IncentiveRewardAm
						let mut v
```

```
                                                if amoun
                                                        \
                                                        S



                                                        ]
                                                }

                                                if v.is_
                                                        *
                                                } else {
                                                        *
                                                }
                                        });
                                }
                        }
                        Ok(())
                }
```

The same idea can also be applied to this `update_claim_reward_deduction_rates()` function.

Impact

Unnecessary code execution, flawed implementation.

Recommended Mitigation Steps

As a rule of thumb, all setters should always have equality checkers as passing an equal to the already stored value hints a mistake and maybe this attempt to add/remove was meant for a different pool.

[04] `OnUpdateLoan::happened()` should check for when `adjustment == 0`

Take a look here.

```
fn happened(info: &(T::AccountId, CurrencyId, Amo
        let (who, currency_id, adjustment, _prev
        let adjustment_abs = TryInto::<Balance>::

        if adjustment.is_positive() {
                <orml_rewards::Pallet<T>>::add_sh
        } else {
                <orml_rewards::Pallet<T>>::remove
        };
}
```

This function has been used to settle adjustments in protocol. The main issue about this report is the fact that adding/removing the share is dependent on if the adjustment is positive or not. This is a somewhat flawed implementation, as the `is_positive()` getter would actually return `false` for when `adjustment == 0` and as such lead to an unnecessary attempt of removing `0` shares.

Impact

Not enough input validation is applied, not the best code structure.

Recommended Mitigation Steps

Consider checking in the function to ensure that for instances where `adjustment == 0` none of neither `add_shares()` or `remove_shares()` is called.

[05] Protocol does not apply deadlines when dealing with critical operations

Consider the code here.

```
		fn do_deposit_dex_share(who: &T::AccountId, lp_cu
			ensure!(lp_currency_id.is_dex_share_curre

			T::Currency::transfer(lp_currency_id, who
			<orml_rewards::Pallet<T>>::add_share(who,

			Self::deposit_event(Event::DepositDexShar
				who: who.clone(),
				dex_share_type: lp_currency_id,
				deposit: amount,
			});
			Ok(())
		}

		fn do_withdraw_dex_share(who: &T::AccountId, lp_c
			ensure!(lp_currency_id.is_dex_share_curre
			ensure!(
				<orml_rewards::Pallet<T>>::shares
				Error::<T>::NotEnough,
			);

			T::Currency::transfer(lp_currency_id, &Se
			<orml_rewards::Pallet<T>>::remove_share(w

			Self::deposit_event(Event::WithdrawDexSha
				who: who.clone(),
				dex_share_type: lp_currency_id,
				withdraw: amount,
			});
			Ok(())
		}
	}
```

Both functions are used to either deposit/withdraw dex shares from the lps by passing in the currency which is in short just like a swap, case is that this function does not include any deadline protection whatsoever. The transaction could be left hanging for a long time and end up being executed in unfavorable situations.

Impact

This is a pretty popular bug case, where no deadlines have been applied and could lead to users losing funds, sometimes in `$USD` equivalent since their transactions might execute under less favorable conditions than intended.

Recommended Mitigation Steps

Consider requesting a deadline to which a user would like their transaction to be hanging for, and ensure that if the deadline has passed the function does not get executed.

[06] Fix typos

Take a look here.

```
          /// Start unbonding tokens up to `amount`
          /// If bonded amount is less than `amount
          /// unbonding. Token will finish unbondir
          #[pallet::call_index(1)]
          #[pallet::weight(T::WeightInfo::unbond()
          pub fn unbond(origin: OriginFor<T>, #[pa
      //..sip
```

The sentence: “Token will finish unbonding after `UnbondingPeriod` blocks.” should instead be:

”**Tokens** will finish unbonding after `UnbondingPeriod` blocks.”

Another instance can be seen here.

```
/// Claim all available multi currencies
///
/// The dispatch origin of this call must
///
/// - `pool_id`: pool type
#[pallet::call_index(2)]
#[pallet::weight(<T as Config>::WeightInf
pub fn claim_rewards(origin: OriginFor<T>
        let who = ensure_signed(origin)?;

        Self::do_claim_rewards(who, pool_
}
```

The first line should be “Claim all **avalaible** multi currencies rewards for specific `PoolId`.” instead.

[07] Consider using BST instead

Take a look here.

```
pub struct PoolInfo<Share: HasCompact, Balance: HasCompac
        /// Total shares amount
        pub total_shares: Share,
        /// Reward infos <reward_currency, (total_reward,
        pub rewards: BTreeMap<CurrencyId, (Balance, Balar
}
```

Evidently, the reward infos are stores in a B-Tree map which is in short an ordered map based on a `B-Tree`.

`B-Trees` represent a fundamental compromise between cache-efficiency and actually minimizing the amount of work performed in a search. In theory, a binary search tree (BST) is the optimal choice for a sorted map, as a perfectly balanced BST performs the theoretical minimum amount of comparisons necessary to find an element (log2n).

Impact

Non-optimized method of finding elements.

Recommended Mitigation Steps

Consider using binary search trees instead.

[08] Consider not switching off important clippy protection methods

Take a look here.

```
#![cfg_attr(not(feature = "std"), no_std)]
#![allow(clippy::unused_unit)]
#![allow(clippy::too_many_arguments)]
```

Protocol disallows clippy to run the auto tool to warn about unused units or too many arguments; whereas too many arguments could be understandable since it's for the `claim_one()` function here. A justification can't be made for not wanting to use `clippy::unused_unit`.

Impact

Bad code structure.

Recommended Mitigation Steps

In most cases impleemntations that have unused units are mostly flawed and should be sorted out.

Lambda (judge) commented:

> Some of the other ones are opinionated, but not necessarily invalid.

xlc (Acala) disputed and commented:

> [01], [05] and [07] are completely invalid. Others are super minor.

> [01] - is 100% not an issue. This is how Substrate pallet works.
> [05] - doesn't make sense. There is no swap happening. The LP tokens are minted already and this only stake the already minted token.
> [07] - doesn't make sense and wouldn't make any difference anyway for the given amount of expected items. It is a premature optimization.

Audit Analysis

For this audit, 6 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **DarkTower** received the top score from the judge.

The following wardens also submitted reports: hunter_w3b, 0xepley, LinKenji, Bauchibred, and ZanyBonzy.

Overview

"Building the liquidity layer of Web3 finance".

Acala, at it's core is a crosschain DeFi network liquidity hub built for

Polkadot. It is an Ethereum compatible, protocol that consists of a decentralized finance network and liquidity hub for Polkadot. Acala's whole infrastructure also includes a stablecoin network and liquidity staking platform.

The codebase presented for this interaction is the Acala staking and reward system. Its review of staking, reward accumulation and distribution as well as bonding/locking of the staking factor are in-scope for this analysis review and will be delved into deeper.

For this Analysis report, we focused mainly on the staking module.

Architecture Overview

During the course of our engagement for conducting this analysis of the staking, bonding and earning modules, we strived to understand the architecture/mechanics of the codebase better first. A summarization of each architecture is as follows:

1. **Pools:** These facilitate staking in the Acala ecosystem. Pools attract incentives for the stakers in such pool at every given period. These incentives can be claimed at any time - with or without a penalty applied.
2. **Rewards:** These get paid out to the users and can be referred to as incentives as we mentioned above. But without pools, there are no incentives. What's the point of rewards for stakers when there are zero stakers? None. Zero point. Rewarding is set aside for legitimate stakers. There are no rewards for stakers with `0` factors in pools. Each stake accrues rewards proportional to their staking factor (weight of stake in the pool for the particular user).
3. **Bonding:** Users typically bond their stakes for a period of time. When that period has elapsed, they can unbond their stake and then

withdraw it from pools. This is where the penalty we mentioned in Pools earlier comes in. If a user bonds for 30 days and decides to unbond on day 10. They can bond but only if they forfeit the penalty fee relative to their stake factor.

Approach Taken in Evaluating Acala

Before we delve in, it was crucial to understand what each module is doing on a distinct level, i.e. as a separate package, how each module integrates with the other module. The modules in scope for this audit are three so we took to understanding each one separately.

We focused mainly on each module's implementation, safeguards, and integration with other modules. With the limited time (we only spent ~1hr each day for 7 days) we had to look into this codebase, we spent the most of our time covering the above mentioned logic Acala.

We added some inline comments on code snippets we cover in this Analysis report to provide context for the reader.

Days 1-2:

- Getting context on Acala from the provided docs: Docs
- Grasping code implementations of staking entry and exit points.

Days 2-4

- Brainstorm some attack vectors relevant to doubling rewards, DoS of rewards & staking functions.
- Set out building a test suite from scratch.
- Further reviews of the staking, earning and bonding modules.

Days 4-7

- Delving deeper into the codebase's test suite.
- Draft & submission of report.

Acala Modules Analysis

Reward accumulation across Block Initializations

```
#[pallet::hooks]
        impl<T: Config> Hooks<BlockNumberFor<T>> for Pal
                fn on_initialize(now: BlockNumberFor<T>)
                        // accumulate reward periodically
                        if now % T::AccumulatePeriod::get
                                let mut count: u32 = 0;
                                let shutdown = T::Emerge

                                for (pool_id, pool_info)
                                        if !pool_info.to
                                                match po
                                                        /
                                                        F

                                                        ]
                                                        -

                                                        ]
                                                }
                                        }
                                }

                                T::WeightInfo::on_initia
                        } else {
                                Weight::zero()
                        }
                }
        }
```

Acala's staking system has this initialization process per blocks. Let's break down this function to tiny bits in order to understand what's happening in the code blocks:

- `on_initialize` is called during block initialization.
- Checks are in place for ensuring the current block number (`now`) is greater than the base accumulation period (`T::AccumulatePeriod::get()`).
- If point 2 above is true, then we proceed to accumulate rewards by iterating through all pools stored in `orml_rewards::PoolInfos` .
- For every one of those pools, we check if the total shares for such pool is not zero -> `if !pool_info.total_shares.is_zero()` .
- If the pool shares is indeed not 0, we check to see if the protocol is under the emergency shutdown i.e like the pausing mechanism of OZ `(T::EmergencyShutdown::is_shutdown())` .
- If the contract is paused, we log a message indicating the skip of accumulation of incentives for that pool.
- Otherwise, if the contract is not paused, we increment the counter to move to the next pool and we don't forget to call `accumulate_incentives` for that specific pool we just went through.
- Lastly, we return from the function.

User Entry point: Staking

```
pub fn deposit_dex_share(
    origin: OriginFor<T>, // originator of txn
    lp_currency_id: CurrencyId, // lp currency token
    #[pallet::compact] amount: Balance, // amount to LP
  ) -> DispatchResult { // returns a result
    let who = ensure_signed(origin)?; // beneficiary of s
    Self::do_deposit_dex_share(&who, lp_currency_id, amou
    Ok(()) // end of txn call
  }
```

Staking LP tokens is the same as `deposit_dex_share` within the Acala staking system. Let's break down this function:

- Users execute this function to stake LP.
- The origin aka user/address must be `Signed` i.e. only authenticated accounts can execute this function.
- `lp_currency_id` : As we have noted in the comment above, this is the type of LP token being staked.
- `amount` : the amount of the LP tokens to be staked.
- `do_deposit_dex_share` : execute the deposit and allocate the shares to the user.

Going back a bit from this function's execution, let's understand that:

- The `ensure_signed` function call is used to make sure that the origin is signed by a valid account. If that is not the case, the transaction to deposit LP tokens reverts.
- The `do_deposit_dex_share` function is called internally to handle the rest of the deposit operation. Any errors encountered during this function's execution will propagate up also cause the entire transaction to revert.

```
fn do_deposit_dex_share(who: &T::AccountId, lp_currency_i
        ensure!(lp_currency_id.is_dex_share_currency_id()

        T::Currency::transfer(lp_currency_id, who, &Self:
        <orml_rewards::Pallet<T>>::add_share(who, &PoolIc

        Self::deposit_event(Event::DepositDexShare {
                who: who.clone(),
                dex_share_type: lp_currency_id,
                deposit: amount,
        }); // @note emit the deposit event
        Ok(()) // @note return
}
```

- The internal `do_deposit_dex_share` function first ensures that the `lp_currency_id` is a valid DEX share currency ID. This validation is in place to prevent stakers from attempting to deposit shares into invalid or non-existent pools.
- When the input validation is complete and the currency is valid, we transfer the specified amount of LP tokens from the user's account to the contract's account using the `T::Currency::transfer` function call.
- Upon the successful transfer of LP tokens from the caller to us, we then add/mint shares to the user's balance for the specified DEX pool using the `orml_rewards::Pallet<T>::add_share` function call.
- Lastly, we fire the `deposit_event` providing details of the deposit and return from the function call.

User Exit point: Withdraw

```
pub fn withdraw_dex_share(
                origin: OriginFor<T>,
                lp_currency_id: CurrencyId,
                #[pallet::compact] amount: Balance,
        ) -> DispatchResult {
                let who = ensure_signed(origin)?;
                Self::do_withdraw_dex_share(&who, lp_curr
                Ok(())
        }
```

Similar to deposits, withdrawing LP is the same as `withdraw_dex_share`. This function allows stakers to withdraw LP tokens, consequently removing/getting rid of the previously allocated shares of the staker from the Pool.

- Close in relation to the `deposit_dex_share` function, the origin must be `Signed` by the transactor for this transaction call ensuring only authenticated accounts can call and execute withdraws.
- `lp_currency_id`: This also represents the type of LP token from which shares are being withdrawn.
- `amount`: This last parameter is the amount of LP tokens to be withdrawn by the caller.

The internal function `do_withdraw_dex_share` handles most of the execution process for this function as we will see below:

```
fn do_withdraw_dex_share(who: &T::AccountId, lp_currency_
                ensure!(lp_currency_id.is_dex_share_curre
                ensure!(
                        <orml_rewards::Pallet<T>>::shares
                        Error::<T>::NotEnough,
                ); // @note ensure user has enough shares

                T::Currency::transfer(lp_currency_id, &Se
                <orml_rewards::Pallet<T>>::remove_share(w

                Self::deposit_event(Event::WithdrawDexSha
                        who: who.clone(),
                        dex_share_type: lp_currency_id,
                        withdraw: amount,
                });  // @note withdraw event emission
                Ok(())
        }
```

As we can see from the function's implementation and our provided comments, the function basically ensures users attempting to withdraw:

1. Have sufficient shares balance to withdraw from.
2. Receive their LP token and accrued rewards.
3. Proper withdrawal event is emitted.

Reward claims

```
    #[pallet::call_index(2)]
            #[pallet::weight(<T as Config>::WeightInfo::clai
            pub fn claim_rewards(origin: OriginFor<T>, pool_
                    let who = ensure_signed(origin)?; // @no

                    Self::do_claim_rewards(who, pool_id) // @
            }
```

The `claim_rewards` function allows users to claim all available multi-currency rewards for a specific pool that corresponds to the provided `pool_id` argument. Let's break down the implementation further:

- Similar to both deposit and withdraw functions for LP, the dispatch origin for this claim call must be Signed by the transactor. In other words, only authenticated accounts can call this function.
- `origin` : the origin of the transaction, which must be signed by the

user.

- `pool_id` : Specification of the pool for which rewards are being claimed from.

Now, we need to look into the internal `do_claimed_rewards` function to completely grasp the call trace.

```
fn do_claim_rewards(who: T::AccountId, pool_id: PoolId) -
                // orml_rewards will claim rewards for al
                <orml_rewards::Pallet<T>>::claim_rewards(

                PendingMultiRewards::<T>::mutate_exists(p
                        if let Some(pending_multi_rewards
                                let deduction_rate = Self
                                let deduction_currency =

                                for (currency_id, pending
                                        if pending_reward
                                                continue;
                                        }

                                        let deduction_rat
                                                // only a
                                                if deduct
                                                        (
                                                } else {
                                                        Z
                                                }
                                        } else {
                                                // apply
                                                deductior
                                        };

                                        let (payout_amour
                                                let shoul
                                                (
                                                        p
                        S
                )
        };

        // payout reward
        match Self::payou
                pool_id,
                &who,
                *currency
                payout_am
                deduction
        ) {
                Ok(_) =>
                        /
                        *

                        S









                        ]
                }
                Err(e) =>
                        T








                        )
                }
        };
}

// clear zero value item
pending_multi_rewards.ret

// if pending_multi_rewar
if pending_multi_rewards.
```

```
                                    *maybe_pending_mu
                              }
                        }
                  });

                  Ok(())
            }
```

- The function facilitates claiming of rewards for a specific pool by a user.
- Reward claims: We call `orml_rewards::Pallet<T>::claim_rewards` to claim rewards for all currencies in the specified pool. Keep in mind the `orml_rewards` module handles the rewarding logic.

Deduction calculation: For every pending reward in the `PendingMultiRewards` storage, we calculates the deduction amount based on the configured deduction rate (for each user).

- The deduction rate is determined by `claim_reward_deduction_rates` and `ClaimRewardDeductionCurrency::<T>::get(pool_id)`.
- If a deduction currency is specified, the deduction rate is then applied to that currency. Otherwise, it is applied to all currencies.
- The deducted amount is then subtracted from the pending reward, resulting in the actual payout amount.

Reward Payout:

- We then `payout_reward_and_reaccumulate_reward` to distribute the rewards to the staker and re-accumulate the deducted amount into the pool (for the loyalty program Acala enforces).
- If the payout and re-accumulation process is deemed successful, the pending reward is updated to zero, and an event (`Event::ClaimRewards`) is emitted to log the details of the claimed

rewards.

State mutation and storage reset:

- After processing all pending rewards, the function then keeps only non-zero value items in the `PendingMultiRewards` storage.
- If `pending_multi_rewards` is empty indicating that all rewards have been claimed, the storage entry is then cleared.

Reward Deduction Rates

```
pub fn update_claim_reward_deduction_rates(
        origin: OriginFor<T>, // @note txn origin
        updates: Vec<(PoolId, Rate)>, // @note data struc
) -> DispatchResult {
        T::UpdateOrigin::ensure_origin(origin)?; // @note
        for (pool_id, deduction_rate) in updates { // @nc
                if let PoolId::Dex(currency_id) = pool_ic
                        ensure!(currency_id.is_dex_share_
                }
                ClaimRewardDeductionRates::<T>::mutate_ex
                        let mut v = maybe_rate.unwrap_or_
                        if deduction_rate != *v.inner() {
                                v.try_set(deduction_rate)
                                Self::deposit_event(Event
                                        pool: pool_id,
                                        deduction_rate,
                                });
                        }

                        if v.inner().is_zero() {
                                *maybe_rate = None;
                        } else {
                                *maybe_rate = Some(v);
                        }
                        Ok(())
```

```
                    })?;
            }
            Ok(())
    }
```

`update_claim_reward_deduction_rates` allows for updating claim reward deduction rates for all reward currencies of a specific Pool

- `updates` : A somewhat data struct of tuples representing the `PoolId` and corresponding deduction rate to be applied.
- For each `update` in the `updates` vector/data struct, the function then proceeds to check if the `PoolId` corresponds to a valid DEX pool. If that is the case, we ensure that the associated currency ID is a valid DEX share currency ID.
- Moving on, for each `update` , we mutate the `ClaimRewardDeductionRates` storage to update the deduction rate associated with the specified `PoolId` .
- If the new deduction `rate` differs from the current `rate` , it is set, and the `ClaimRewardDeductionRateUpdated` event is emitted to log the update.
- Keep in mind that if the `rate` attempted to be set is the same as the current `rate` , we skip doing the update as it's pointless in that case to set an already set state.
- If the deduction `rate` becomes zero after the update, the corresponding entry in the `ClaimRewardDeductionRates` storage is then removed to conserve storage space.

Call-trace Diagrams

Note: to view the provided images, please see the original submission ***here****.*

Codebase Quality

Here are some of our observations of the `Acala` codebase's quality:

Readability

The protocol's function implementations are well documented and easy to follow. The code blocks in general follow a consistent naming guide that hints at what the function implementation does and is supposed to return after execution.

Maintainability

Each architecture component is separated into modules. For example, the staking module is separate from the reward modules. They each utilize the other' implementation as we can see in the staking module where claiming of rewards borrows code implementation from the rewards code logic.

This effectively makes the codebase manageable as you can change a function implementation in one module without introducing breaking changes in other modules that utilize it.

Performance

Testing suite for the 3 modules in sope are great. They are normal unit-style tests but they live in their separate files away from the production code.

Robustness

In most of the function implementation across the codebase, errors are handled from the onset of the function's execution. This is great. It ensures the call can already be reverted without getting up to the point where the invalid state resulting from the bad data is reached to bubble up the error.

Systematic Risks and Centralization

This section of the analysis report covers error handling, slight flaws and major systemic risks of the covered components in the Acala Staking System.

Pool initialization: `/incentives/src/lib.rs`

Pools are created by Governance in the Acala staking system. So, users are not able to create arbitrary staking pools which is good. But, a malicious governor privileged individual can mass create pools with empty stakes that is zero shares and DoS the entire staking mechanism when we run into reward accumulation and pool iteration during blocks initialization in the code snippet below:

```
    for (pool_id, pool_info) in orml_rewards::PoolInfos::<T>
     if !pool_info.total_shares.is_zero() {
       match pool_id {
         // do not accumulate incentives for PoolId::Loans a
         PoolId::Loans(_) if shutdown => {
           log::debug!(
             target: "incentives",
             "on_initialize: skip accumulate incentives for
             pool_id
           );
         }
         _ => {
           count += 1;
           Self::accumulate_incentives(pool_id);
         }
       }
     }
   }
```

Assuming the Governance code becomes vulnerable to malicious actors, one way to put an end to staking by DoS is mass creation of pools that are not legitimate and force the loop interaction of pools in the next block initialization to fail.

Insights

Pool initialization: `/incentives/src/lib.rs`

The current logic of looping through pools is non scalable. Optimization is completely thrown out the window this way. I'm sure the team believes governance will not be malicious or governance code implementation will always be safe but once any one of these happen to not be the case, the whole staking model rests on the balance. Instead of looping over pools, implement caching or use structs to store necessary pool information and use that during the block initializations. This brings scalability back into the control of the protocol.

In summary, the Acala codebase review was a great learning experience for understanding some niche aspects such as their utilization of block initializations. After having spent a couple days understanding the high level concept, we finally were able to pull up this report for understanding some parts of the protocol.

Time spent

7 hours

xlc (Acala) acknowledged

Note: For full discussion, see here.

Disclosures

Top