Anchor contest
Findings & Analysis Report

2023-01-18

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 contest 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 contest outlined in this document, C4 conducted an analysis of the Anchor smart contract system written in Rust. The audit contest took place between February 24—March 9 2022.

Wardens

20 Wardens contributed reports to the Anchor contest:

1. cmichel
2. WatchPug (jtp and ming)
3. jmak
4. csanuragjain
5. BondiPestControl (leastwood and kirk-baird)
6. defsec
7. hubble (ksk2345 and shri4net)
8. broccoli (shw and jonah1005)
9. hickuphh3

10. gzeon
11. 0xwags
12. robee
13. IllIllI
14. 0xliumin
15. cccz
16. 0v3rf10w

This contest was judged by Albert Chon.

Triage performed by Alex the Entreprenerd.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 15 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 12 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 13 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 8 reports recommending gas optimizations.

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

Scope

The code under review can be found within the C4 Anchor contest repository.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into 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

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#).

High Risk Findings (3)

[H-01] Spend limit on owner can be bypassed

Submitted by csanuragjain, also found by cmichel

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/distributor/src/contract.rs#L140
https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/community/src/contract.rs#L69

It seems that the owner is only allowed to spend amount uptil config.spend_limit. However it was observed that this `config.spend_limit` is never decreased even if owner has spend an amount. This makes `config.spend_limit` useless as owner can simply send 2-multiple transactions each of `config.spend_limit` which will all pass and hence bypassing the spend limit placed on owner.

Proof of Concept

1. Assume spend limit of 100 is placed on owner
2. Owner simply calls the spend function at either distributor or community contract with amount 100
3. Ideally after this transaction owner should not be allowed to perform any more spend operation

4. Since `config.spend_limit` remains unchanged, owner can call step 2 multiple times which will spend amount 100 several times bypassing spend limit

Recommended Mitigation Steps

After successful spend, the `config.spend_limit` should be decreased by the amount spend.

[Albert Chon (judge) commented via duplicate issue #34]:

> Indeed, this is a serious oversight, unless one expects the whitelisted addresses to not exceed the spend limit (which is not a good assumption to bake in).

[H-02] `money-market-contracts/oracle#feed_prices()` delayed transaction may disrupt price feeds

Submitted by WatchPug

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/money-market-contracts/contracts/oracle/src/contract.rs#L106-L113

The implementation only takes two attributes: `asset` and `price`. And the `last_updated_time` of the record will always be set to the current `block.time`.

This makes it possible for the price feeds to be disrupted when the network is congested, or the endpoint is down for a while, or the `feeder` bot handled the message queue inappropriately, as a result, the transactions with stale prices get accepted as fresh prices.

Since the price feeds are essential to the protocol, that can result in users' positions being liquidated wrongfully and case fund loss to users.

Proof of Concept

Given:

- `feeder` i connected to an endpoint currently experiencing degraded performance;
- ETH price is `$10,000`;
- The `max_ltv` ratio of ETH is `60%`.
- Alice borrowed `5,000 USDC` with `1 ETH` as collateral;
- ETH price dropped to `$9,000`, to avoid liquidation, Alice repaid `1,000 USD`;
- The price of ETH dropped to `$8,000`; `feeder` tries to `updateMainFeedData()` with the latest price: `$8,000`, however, since the network is congested, the transactions were not get packed timely;
- ETH price rebound to `$10,000`; Alice borrowed another `1,000 USDC`;
- The txs send by `feeder` at step 3 finally got packed, the protocol now believes the price of ETH has suddenly dropped to `$8,000`, as a result, Alice's position got liquidated.

Recommended Mitigation Steps

Change to:

```
pub fn feed_prices(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    prices: Vec<(String, Decimal256, u64)>,
) -> Result<Response, ContractError> {
    let mut attributes = vec![attr("action", "feed_prices")];
    let sender_raw = deps.api.addr_canonicalize(info.sender.as_s
    for price in prices {
        let asset: String = price.0;
        let mut updated_time: u64 = price.2;
        let price: Decimal256 = price.1;

        // Check feeder permission
        let feeder = read_feeder(deps.storage, &asset)?;
        if feeder != sender_raw {
            return Err(ContractError::Unauthorized {});
        }

        let config: Config = read_config(deps.storage)?;
```

```
            if env.block.time.seconds() > updated_time {
                // reject stale price
                if env.block.time.seconds() - updated_time > config.
                    return Err(ContractError::InvalidInputs {});
                }
            } else {
                // reject future timestamp, graceFuturePeriod can be
                if updated_time - env.block.time.seconds() > config.
                    return Err(ContractError::InvalidInputs {});
                }
                updated_time = env.block.time.seconds();
            }

            attributes.push(attr("asset", asset.to_string()));
            attributes.push(attr("price", price.to_string()));

            store_price(
                deps.storage,
                &asset,
                &PriceInfo {
                    last_updated_time: updated_time,
                    price,
                },
            )?;
        }

        Ok(Response::new().add_attributes(attributes))
    }
```

bitn8 (Anchor) disputed and commented:

> We currently have a mean shorting function that pulls multiple price feeds so that if one is stale it gets rejected.

Alex the Entreprenerd (triage) commented:

> Seems like the warden has shown a specific scenario, contingent on external conditions.

> However, from the code, there seems to be no “mean shorting function”, at least in the code in scope.

Albert Chon (judge) commented:

> Agreed with @Alex the Entreprenerd, oracle staleness is still an issue in this version of the code.

[H-03] Missing Access Control for `FabricateMIRClaim` and `FabricateANCClaim`

Submitted by jmak

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/anchor-bAsset-contracts/contracts/anchor_airdrop_registry/src/contract.rs#L109
https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/anchor-bAsset-contracts/contracts/anchor_airdrop_registry/src/contract.rs#L71

`FabricateMIRClaim` and `FabricateANCClaim` should only be issued by the Hub contract (the central hub for all minted bLuna managed by Lido). However, `execute_fabricate_anchor_claim` and `execute_fabricate_mir_claim` do not restrict the caller, allowing anyone to submit these msgs.

Recommended Mitigation Steps

Recommended to add at least simple access control checks in the contract to ensure that these functions can only be called by the Hub and not by others.

See a below for a potential code snippet.

```
    // only hub contract can send this message.
    let config = read_config(deps.storage)?;
    let sender_raw = deps.api.addr_canonicalize(&info.sender.to_stri
    if sender_raw != config.hub_contract {
        return Err(StdError::generic_err("unauthorized"));
    }
```

Alex the Entreprenerd (triage) commented:

> The finding is correct in that anyone can call the function.

> The finding lacks any form of impact as to what would happen.

> I think these may be test functions also.

> Not convinced the report makes sense for high severity given the lack of detail.

Medium Risk Findings (12)

[M-01] When transferring tokens not in `whitelist` on Ethereum to Terra with `CrossAnchorBridge.depositStable()` , the funds may get frozen

Submitted by WatchPug, also found by BondiPestControl, broccoli, and defsec

In the current implementation of `CrossAnchorBridge`, all `require` that "Check that `token` is a whitelisted token" is commented out.

As a result, users may send transcations with the non-whitelisted tokens and as they can not be processd properly on the Terra side, the funds can be frozen on the brige contract or on the Terra side.

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/cross-chain-contracts/ethereum/CrossAnchorBridge.sol#L167-L175

```
    function handleStableToken(
            address token,
            uint256 amount,
            uint8 opCode
        ) internal {
            // Check that `token` is a whitelisted stablecoin token.
            // require(whitelistedStableTokens[token]);
            handleToken(token, amount, opCode);
    }
```

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/cross-chain-contracts/ethereum/CrossAnchorBridge.sol#L250-L253

```
function redeemStable(address token, uint256 amount) external {
        // require(whitelistedAnchorStableTokens[token]);
        handleToken(token, amount, OP_CODE_REDEEM_STABLE);
    }
```

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/cross-chain-contracts/ethereum/CrossAnchorBridge.sol#L255-L258

```
function lockCollateral(address token, uint256 amount) external
        // require(whitelistedCollateralTokens[token]);
        handleToken(token, amount, OP_CODE_LOCK_COLLATERAL);
    }
```

Recommended Mitigation Steps

Uncomment the whitelist codes.

bitn8 (Anchor) disagreed with severity

Alex the Entreprenerd (triage) commented:

> Seems conditional, would recommend downgrading to Medium.

Albert Chon (judge) decreased severity to Medium

[M-02] `bEth` Rewards May Be Depleted By Flashloans or Whales

Submitted by BondiPestControl, also found by cmichel https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/anchor-bEth-contracts/contracts/anchor_beth_reward/src/user.rs#L187-L212

Rewards are dispersed to users as a percentage of the user's balance vs total balance (of `bEth`). Rewards are accumulated each time a user calls `execute_decrease_balance()`, `execute_increase_balance()` or `execute_claim_rewards()` as these functions will in term call `update_global_index()` before any balance adjustments.

There is an attack vector where a user may arbitrarily increase their `bEth` balance either by flashloan of purchase of wormhole eth, then converting it bEth via `anchor_beth_converter`. The attacker may then have a significant portion of the total balance. This use can then call the contract which pushes rewards to `anchor_beth_reward` this will increase the `reward_balance` of the contract. Since the attacker controls a large portion of the bEth balance they will receive a higher portion of the rewards. They may then convert the bEth back to wormhole eth and close their position.

The result is the user has not contributed anything to the system but still gained a high portion of the rewards.

Proof of Concept

The logic for `update_global_index()` does not account for users suddenly increasing the balance.

```
fn update_global_index(state: &mut State, reward_balance: Uint12
    // Zero staking balance check
    if state.total_balance.is_zero() {
        // nothing balance, skip update
        return Ok(());
    }

    // No change check
    if state.prev_reward_balance == reward_balance {
        // balance didnt change, skip update
        return Ok(());
    }
```

```
        // claimed_rewards = current_balance - prev_balance;
        let claimed_rewards = reward_balance.checked_sub(state.prev_

        // update state
        state.prev_reward_balance = reward_balance;
        // global_index += claimed_rewards / total_balance;
        state.global_index = decimal_summation_in_256(
            state.global_index,
            Decimal::from_ratio(claimed_rewards, state.total_balance
        );

        Ok(())
    }
```

Steps:
a) Flashloan (or buy) a significant portion of wormhole Eth
b) Convert to Anchor bEth via `anchor_beth_converter`
c) Push rewards on `anchor_beth_rewards`
d) `anchor_beth_rewards::execute_claim_rewards()`
e) Convert Anchor bEth to wormhole Eth
f) Repay flashloan (or sell wormhole eth)

Recommended Mitigation Steps

There are multiple possible mitigations to this issue.

First, option is to only allow the `global_index` to be updated once per block. In addition to this, cap the amount of rewards that may be paid per block (keeping the remaining rewards for the next block). This would reduce the effectiveness of the attack and limit the amount they may earn per block.

Another option is to induce a wait time before the user may begin earning rewards. However, this would require a second transaction from the user to begin collection their reward which may hurt UX.

[M-03] Governance Voting Dis-proportionally Favours Users

Who Stake And Vote After A Poll Has Been Created And Had Its Snapshot Taken

Submitted by BondiPestControl

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/gov/src/contract.rs#L543-L580
https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/gov/src/contract.rs#L582-L665
https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/gov/src/staking.rs#L15-L57
https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/gov/src/contract.rs#L364-L455

Polls are created by targeting the `receive_cw20` function which is queried whenever the contract receives tokens. By setting the hook message to `Cw20HookMsg::CreatePoll`, the sender is able to create a poll, assuming the amount sent satisfies the minimum deposit amount for poll creation. Users can also choose to call `ExecuteMsg::SnapshotPoll` or have it handled automatically when a user casts a vote on the newly created poll.

The snapshot simply sets `a_poll.staked_amount`, which represents the total staked amount within the governance contract at a given block. However, during the voting period, other users can stake tokens and effectively have an increasing influence over the outcome of a given poll. There are no check-pointed balances to ensure that a certain user had staked tokens at the time the poll had its snapshot taken.

This can be abused to skew poll results in favour of users who stake their Anchor tokens after a poll has had its snapshot taken.

Proof of Concept

Let's assume the share to token exchange rate is `1:1` such that if a user deposits 100 Anchor tokens, they receive 100 shares in return.

Consider the following scenario:

- There are a total of 100 Anchor tokens in the Governance contract.

- Alice creates a poll and executes `ExecuteMsg::SnapshotPoll` such that `a_poll.staked_amount == 100`.
- Bob deposits 10 Anchor tokens through the `Cw20HookMsg::StakeVotingTokens` hook message which increases the contract's total balance to 110 and shares to 110 as the exchange rate is `1:1` upon minting and redeeming shares.
- At this point, the target poll has a `a_poll.staked_amount == 100`, even though there are really 110 Anchor tokens staked.
- As a result, if Bob votes on a poll, they have a 10% degree of influence on the outcome of the poll, even though they have less than 10% of the total staked tokens (i.e. 10/110).
- Therefore, poll voters are actually incentivised to stake tokens after a poll has had its snapshot taken in order to maximise their voting power.

Recommended Mitigation Steps

Consider implementing a check-pointing mechanism such that when a user casts a vote, the user's staked balance is checked at the block height upon which the snapshot was taken instead of checking its most up-to-date staked balance. This check-pointing behaviour is implemented on Ethereum which has a more restrictive block space. The mechanism will simply store the staker's balance on each stake/unstake action. When user's wish to vote, the protocol will check the balance at a specific block (i.e. the snapshotted block). An example implementation can be found here.

bitn8 (Anchor) disagreed with severity and commented:

> I wouldn't call this a critical bug. Nevertheless, this will be addressed with ve-ANC tokenomics where tokens have to lock for periods of time (curve lock tokenomics).

Alex the Entreprenerd (triage) commented:

> Loss of Yield = Medium seems appropriate

[M-04] Sandwich attack on astroport sweep

Submitted by cmichel

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts%2Fanchor-token-contracts%2Fcontracts%2Fcollector%2Fsrc%2Fcontract.rs#L130-L137

The collector contract allows anyone to `sweep`, swapping an asset token to ANC through astro port.
Note that `belief_price` is not set and `config.max_spread` might not be set as well or misconfigured.

This allows an attacker to create a contract to perform a sandwich attack to make a profit on this trade.

> A common attack in DeFi is the sandwich attack. Upon observing a trade of asset X for asset Y, an attacker frontruns the victim trade by also buying asset Y, lets the victim execute the trade, and then backruns (executes after) the victim by trading back the amount gained in the first trade. Intuitively, one uses the knowledge that someone's going to buy an asset, and that this trade will increase its price, to make a profit. The attacker's plan is to buy this asset cheap, let the victim buy at an increased price, and then sell the received amount again at a higher price afterwards.

Trades can happen at a bad price and lead to receiving fewer tokens than at a fair market price.
The attacker's profit is the protocol's loss.

Proof of Concept

Attacker creates a contract that triggers 3 messages for the sandwich attack:

- Astroport: buy ANC with asset
- Call `sweep` which trades at bad price
- Astroport: sell assets from the first message for profit

Recommended Mitigation Steps

Consider setting a ANC/asset `belief_price` from an oracle.

bitn8 (Anchor) disagreed with severity and commented:

> While it might certainly be possible to sandwich an attack like this, there is no risk to users' funds outside a possible bad fill price. There are many other issues at work here too such as UNI v2 that Astro is built on and volatility. This assumes very illiquid markets or huge wallets to move prices like this.

Alex the Entreprenerd (triage) commented:

- Anyone can call = can set up a price imbalance
- Is a sweep (prob dust amount)

> Looks valid but impact seems reduced.

[M-05] `anchor_basset_reward` pending yields can be stolen

Submitted by WatchPug, also found by csanuragjain

For yield farming aggregators, if the pending yield on an underlying strategy can be harvested and cause a surge of rewards to all existing investors, especially if the harvest can be triggered permissionlessly. Then the attacker can amplify the attack using a flash loan.

This is a well-known attack vector on Ethereum.

The root cause for this attack vector is that the pending yield is not settled to the existing users before issuing shares to new deposits.

In the current implementation of anchor_basset_reward/src/user.rs#execute_increase_balance() before L105, the `state.global_index` is not being upadted first.

- Expected: the pending rewards (current_global_index - state.global_index) belongs to old user balance,
- Current implementation: the pending rewards (current_global_index - state.global_index) will be multiplied by the new user balance after `increase_balance` when calculate rewards.

Because new user balance > old user balance, the user will take a part of the rewards belonging to other existing users.

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/anchor-bAsset-contracts/contracts/anchor_basset_reward/src/user.rs#L80-L123

```
pub fn execute_increase_balance(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    address: String,
    amount: Uint128,
) -> StdResult<Response<TerraMsgWrapper>> {
    let config = read_config(deps.storage)?;
    let owner_human = deps.api.addr_humanize(&config.hub_contrac
    let address_raw = deps.api.addr_canonicalize(&address)?;
    let sender = info.sender;

    let token_address = deps
        .api
        .addr_humanize(&query_token_contract(deps.as_ref(), owne

    // Check sender is token contract
    if sender != token_address {
        return Err(StdError::generic_err("unauthorized"));
    }

    let mut state: State = read_state(deps.storage)?;
    let mut holder: Holder = read_holder(deps.storage, &address_

    // get decimals
    let rewards = calculate_decimal_rewards(state.global_index,

    holder.index = state.global_index;
    holder.pending_rewards = decimal_summation_in_256(rewards, h
    holder.balance += amount;
    state.total_balance += amount;

    store_holder(deps.storage, &address_raw, &holder)?;
    store_state(deps.storage, &state)?;

    let attributes = vec![
        attr("action", "increase_balance"),
        attr("holder_address", address),
        attr("amount", amount),
    ];

    let res = Response::new().add_attributes(attributes);
    Ok(res)
}
```

Proof of Concept

Given:

- the reward balance of `anchor_basset_reward` increased

The attacker can:

1. `bond` a large amount of asset tokens

```
   -   [anchor_basset_hub](https://github.com/code-423n4/2022-02-ar
   -   [anchor_basset_token](https://github.com/code-423n4/2022-02-
   -   [anchor_basset_reward](https://github.com/code-423n4/2022-02
```

2. `UpdateGlobalIndex` on anchor_basset_hub

   - `anchor_basset_hub` will trigger anchor_basset_reward's `execute_update_global_index()` and increase `global_index` for all users

As of now, the attacker can get a large share of the pending yield. The attacker can claim the rewards and exit.

This process can be done in one transaction by using a smart contract, and the impact can be amplified by using a flash loan.

Recommended Mitigation Steps

Consider changing to a similar approach like `anchor_beth_reward/src/user.rs#L114`, update `state.global_index` before changing the user's balance.

And/or, transfer rewards and update `global_index` in one transaction.

Alex the Entreprenerd (triage) commented:

> Seems like loss of yield due to frontrun.

Albert Chon (judge) decreased severity to Medium and commented:

> Indeed a bug, although this bug can be mitigated in practice by continuously executing `UpdateGlobalIndex`, doing so shouldn't necessarily be a requirement for the functioning of the system.

[M-06] Simple interest calculation is not exact

Submitted by cmichel

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts%2Fmoney-market-contracts%2Fcontracts%2Fmarket%2Fsrc%2Fborrow.rs#L304

The borrow rate uses a simple interest formula to compute the accrued debt, instead of a compounding formula.

```
pub fn compute_interest_raw(
    state: &mut State,
    block_height: u64,
    balance: Uint256,
    aterra_supply: Uint256,
    borrow_rate: Decimal256,
    target_deposit_rate: Decimal256,
) {
  // @audit simple interest
    let passed_blocks = Decimal256::from_uint256(block_height -

    let interest_factor = passed_blocks * borrow_rate;
    let interest_accrued = state.total_liabilities * interest_fa
    // ...
}
```

This means the actual borrow rate and interest for suppliers depend on how often updates are made.
This difference should be negligible in highly active markets, but it could lead to a lower borrow rate in low-activity markets, leading to suppliers losing out on interest.

Recommended Mitigation Steps

Ensure that the markets are accrued regularly, or switch to a compound interest formula (which has a higher computational cost due to exponentiation, but can be approximated, see Aave).

Alex the Entreprenerd (triage) commented:

> Without Code and explanation, I'm skeptical of Med (loss of yield), as it could just be dust amount, provided that interest is compounded roughly 1 per month or similar (see compound interest math, and how `e` limits max autocompounds to dust variation)

> Either way, the observation is correct.

[M-07] Updating the hub's token contract address may lead to incorrect undelegation amount

Submitted by hubble

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/anchor-bAsset-contracts/contracts/anchor_basset_hub/src/config.rs#L90-L97

The hub contract allows config updates to the `token_contract` config values in `anchor-bAsset-contracts/contracts/anchor_basset_hub/src/config.rs`.
Such updates can cause wrong amounts of tokens to be calculated during processing of undelegations, since the amount of unbonded bLuna tokens is stored for batched unbonding as `requested_with_fee`.

Proof of Concept

Contract : anchor-bAsset-contracts/contracts/anchor_basset_hub/src/config.rs
Function : pub fn execute_update_config(...)
Line 90 :

```
        if let Some(token) = token_contract {
            let token_raw = deps.api.addr_canonicalize(token.as_str

            CONFIG.update(deps.storage, |mut last_config| -> StdResu
                last_config.token_contract = Some(token_raw);
                Ok(last_config)
            })?;
        }
```

Recommended Mitigation Steps

Its recommended to remove the ability to update `token_contract` config value, or asserting that `requested_with_fee` is zero before allowing an update of the `token_contract` address.

Alex the Entreprenerd (triage) commented:

> Looks like Admin Privilege so Medium seems appropriate.

[M-08] Unbonding validator random selection can be predicted

Submitted by cmichel

When unbonding, the `pick_validator` function is supposed to choose a random validator to unstake from.
However, this randomness can be predicted knowing the block height which is very easy to predict.

```
    let mut iteration_index = 0;

    while claimed.u128() > 0 {
        let mut rng = XorShiftRng::seed_from_u64(block_height + iter
        let random_index = rng.gen_range(0, deletable_delegations.le
        // ...
    }
```

As validators are paid rewards proportional to their stake, choosing where to bond and unbond from directly impacts validator rewards.
Combined with being able to choose the validator when bonding (see `execute_bond`) a validator can steal rewards and increase their own.

Proof of Concept

1. Validator A wishes to increase their rewards. They call `execute_bond(, validator = A)` with their validator.
2. Validator A precomputes the randomness for unbonding if the block was mined at the next block height. If it's a different validator, they call `unbond`. The stake is removed from some other validator and the user receives it back. Note that validator A's stake did not decrease and contains the initial bond amount.
3. They repeat this process until every single bond of the Anchor platform is now staked to validator A and they earn huge rewards while everyone else does not earn anything from Anchor bonds anymore.

Recommended Mitigation Steps

An `unbond` must always remove stake from the validator the stake was originally bonded to, otherwise, it's exploitable by the attack above.
Consider keeping track of all bonds with a `bond_id` and create a map of `bond_id -> validator`.
If a validator's stake decreased due to slashing, take the remaining unstake amount proportionally from all other validators.

bitn8 (Anchor) disagreed with severity

Alex the Entreprenerd (triage) commented:

> Looks like weak RNG, Medium or High seem proper.

Albert Chon (judge) decreased severity to Medium and commented:

> Indeed, block height should definitely not be used as a source of randomness.

[M-09] Potential lock of rewards in the custody contracts

Submitted by broccoli

The `swap_to_stable_denom` function in the custody contracts swaps all other native tokens into a specific one. The function creates swap messages for all the other native tokens and adds them as sub-messages, and handles the reply only when the last sub-message succeeds. Upon receiving the reply, the contract sends the swapped tokens (i.e., rewards) to the overseer contract.

In cases where the last sub-message fails, the custody contract will not receive a reply, and therefore the rewards are left in the contract. The rewards are locked in the contract until someone triggers `swap_to_stable_denom` again, and the last swap succeeds. However, if the last swap consistently fails in some period for any reason, the total rewards will be locked in the contract during that period. As a result, users cannot get the rewards they are supposed to receive in that period.

Proof of Concept

Referenced code:

[`custody_beth/src/distribution.rs#L110-L115`]

[`custody_bluna/src/distribution.rs#L109-L114`]

Recommended Mitigation Steps

Consider handling the reply on either success or failure, i.e., using `ReplyOn::Always`, to avoid the failure of the swap to cause tokens to be locked.

[bitn8 (Anchor) disagreed with severity]

[Alex the Entreprenerd (triage) commented]:

> Reliant on external conditions. Severity seems appropriate.

[M-10] Possible Wrong bAsset Rewards/Borrow limits Calculation

Submitted by defsec

During the code review, It has been observed that reward calculation has been done with execute_epoch_operations function. However, the config are stored in the storage. When the anc_purchase_factor is updated by the owner, the execute_epoch_operations is not called.

Proof of Concept

1. Navigate to the following contract.

```
192 https:
```

2. Config is updated with update_config function.

```
        if let Some(threshold_deposit_rate) = threshold_deposit_rate
            config.threshold_deposit_rate = threshold_deposit_rate;
        }

        if let Some(buffer_distribution_factor) = buffer_distributic
            config.buffer_distribution_factor = buffer_distribution_
        }

        if let Some(anc_purchase_factor) = anc_purchase_factor {
            config.anc_purchase_factor = anc_purchase_factor;
        }

        if let Some(target_deposit_rate) = target_deposit_rate {
            config.target_deposit_rate = target_deposit_rate;
        }

        if let Some(epoch_period) = epoch_period {
            config.epoch_period = epoch_period;
        }

        if let Some(price_timeframe) = price_timeframe {
            config.price_timeframe = price_timeframe;
        }
```

3. After the update, execute_epoch_operations function is not called. That will cause to out-of-date data.

Recommended Mitigation Steps

Consider calling execute_epoch_operations function after config update.

Alex the Entreprenerd (triage) commented:

> Confirmed Lack of validation.

Albert Chon (judge) commented:

> Good catch.

[M-11] `money-market-contracts/contracts/market` `claim_rewards` may revert due to `spend_limit` set on `distributor`

Submitted by WatchPug

While `claim_rewards` from the `money-market`, it calls the `distributor_contract#spend()` to send the rewards.

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/money-market-contracts/contracts/market/src/borrow.rs#L216-L234

```
    let messages: Vec<CosmosMsg> = if !claim_amount.is_zero() {
        vec![CosmosMsg::Wasm(WasmMsg::Execute {
            contract_addr: deps
                .api
                .addr_humanize(&config.distributor_contract)?
                .to_string(),
            funds: vec![],
            msg: to_binary(&FaucetExecuteMsg::Spend {
                recipient: if let Some(to) = to {
                    to.to_string()
                } else {
                    borrower.to_string()
                },
                amount: claim_amount.into(),
            })?,
        })]
    } else {
        vec![]
    };
```

However, the `distributor_contract#spend()` function have a `spend_limit` config and it will revert if the amount is larger than the `spend_limit`.

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/anchor-token-contracts/contracts/distributor/src/contract.rs#L153-L155

```
    if config.spend_limit < amount {
        return Err(StdError::generic_err("Cannot spend more than spe
    }
```

As a result, users won't be able to claim their rewards anymore once the amount of the rewards excess the `spend_limit` config on `distributor_contract`.

Recommended Mitigation Steps

Consider removing the `spend_limit` or allowing users to specify an amount when `claim_rewards`.

Alex the Entreprenerd (triage) commented:

> Loss of yield, conditional on reaching limit, consider reducing severity.

Albert Chon (judge) decreased severity to Medium and commented:

> Can be mitigated with a large spend limit, which likely exists to prevent catastrophic cases. The issue is correct though, that this is a bug. Downgrading to medium severity though, given the practicality.

[M-12] Staking tokens can be stolen

Submitted by cmichel

https://github.com/code-423n4/2022-02-anchor/blob/7af353e3234837979a19ddc8093dc9ad3c63ab6b/contracts/anchor-token-contracts/contracts/gov/src/staking.rs#L88

The staking contract keeps track of *shares* of each user.
When withdrawing from the staking contract the `amount` parameter is converted to `shares` and this value is decreased (`shares = amount / total_balance * total_share`).
This shares calculation rounds down which allows stealing tokens.

The current code seems to try to protect against this by doing a maximum `std::cmp::max(v.multiply_ratio(total_share, total_balance).u128(), 1u128)` but this only works for the case where the shares would round down to zero. It's still exploitable.

```
    let withdraw_share = amount
    // @audit only rounds up when it'd be zero, but should always rc
        .map(|v| std::cmp::max(v.multiply_ratio(total_share, total_k
    // @audit sets it to the `amount` parameter if not `None`
    let withdraw_amount = amount
        .map(|v| v.u128())
```

Proof of Concept

Assume `total_balance = 2e18` and `total_share=1e18`. Then withdrawing/burning one share should allow withdrawing `total_balance / total_share = 2` amount.
However, imagine someone owns 1 shares (bought for 2 amount). Then `withdraw_voting_tokens(amount=3)`. And `withdraw_share = std::cmp::max(v.multiply_ratio(total_share, total_balance).u128(), 1u128) = max(amount=3 * total_share=1e18 / total_balance=2e18, 1) =`

`max("3/2", 1) = 1`
The attacker made a profit.

The attacker can make a profit by staking the least `amount` to receive one share, and then immediately withdrawing this one share by specifying the largest `amount` such that the integer rounding still rounds to this one share. They make a small profit. They can repeat this many times in a single transaction with many messages. Depending on the token price, this can be profitable as similar attacks show.

Recommended Mitigation Steps

The protocol tries to protect against this attack but the `max(., 1)` is not enough mitigation. There are several ways to fix this:

1. Always round up in `withdraw_share`
2. Always recompute the `withdraw_amount` with the new `withdraw_share` even if the `amount` parameter was set.
3. Use a `share` parameter (instead of the `amount` parameter) and use it as `withdraw_share`. Rounding down on the resulting `withdraw_amount` is bad for the attacker as burning a share leads to fewer tokens.

bitn8 (Anchor) disagreed with severity

Albert Chon (judge) decreased severity to Medium and commented:

> The amount would be miniscule due to the initial share amount scale being equal to the staked tokens scale (1e6), but indeed, the attack could be applied infinite times, though it's likely then that the gas costs incurred would exceed the profits gained.

Low Risk and Non-Critical Issues

For this contest, 13 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **hickuphh3** received the top score from the judge.

The following wardens also submitted reports: cmichel, hubble, 0xwags, defsec, 0xliumin, broccoli, cccz, IllIllI, WatchPug, robee, gzeon, *and* BondiPestControl.

Codebase Impressions & Summary

This audit contest is probably the largest in codebase size to date, consisting of various components such as tokens, airdrop, governance, staking, liquidity mining, reward distributions, a money market and cross-chain bridge functionality. While these components aren't complex when reviewed in isolation, the code complexity becomes a little higher because of the various interactions with each other.

Overall, the codebase quality is very high. Inline comments provided are helpful in describing why some checks were not performed (mainly because they are done elsewhere).

The level of documentation is also high, where the documentation portal and READMEs are adequate and kept up-to-date with the codebase.

Tests ran without issues, and could be easily modified to test out POCs and potential vulnerabilities.

The findings made involve the lack of sanity checks for some key config values (eg. their values should not exceed 100%). While the likelihood of the creation of governance polls to modify these values is low because the proposer loses his ANC deposit if quorum isn't met, the impact of such polls, should they be successful, is high.

[L-01] CrossAnchorBridge: Decide if whitelisting feature is to be kept or removed

Line References

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/cross-chain-contracts/ethereum/CrossAnchorBridge.sol#L125-L130

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/cross-chain-contracts/ethereum/CrossAnchorBridge.sol#L147-L151 https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/cross-chain-contracts/ethereum/CrossAnchorBridge.sol#L172-L173

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/cross-chain-contracts/ethereum/CrossAnchorBridge.sol#L251

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/cross-chain-contracts/ethereum/CrossAnchorBridge.sol#L256

Description

The whitelisted mappings are defined and set to `true` for certain token addresses, but the lines of code where they are used are commented out.

Recommended Mitigation Steps

Decide whether to keep the whitelisting requirement. Either comment out the remaining lines / remove them entirely, or uncomment the lines where they are used.

[L-02] Incorrect opcode specification documentation

Line References

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/cross-chain-contracts/README.md#op-code-specification

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/cross-chain-contracts/terra/contracts/wormhole-bridge/src/contract.rs#L99-L125

Description

It is unclear if the full opcode values include the flags as well. If so, they should be 8 bits in length, which isn't the case. Their decimal representations are also incorrect.

```
| Opcode | Flags | Full Opcode | Decimal |
|:-------|:------|:------------|:-------|
| Deposit Stable | `FLAG_BOTH_TRANSFERS` | `0b110000` | `192` |
| Redeem Stable | `FLAG_BOTH_TRANSFERS` | `0b110001` | `193` |
| Repay Stable | `FLAG_INCOMING_TRANSFER` | `0b1000000` | `64` |
| Lock Collateral | `FLAG_INCOMING_TRANSFER` | `0b1000001` | `65
| Unlock Collateral | `FLAG_OUTGOING_TRANSFER` | `0b0100000` | `
| Borrow Stable | `FlAG_OUTGOING_TRANSFER` | `0b0100001` | `33`
| Claim Rewards | `FLAG_OUTGOING_TRANSFER` | `0b0100010` | `34`
```

The opcodes were also not updated in the comments of the wormhole bridge terra contract.

Recommended Mitigation Steps

Update the comments to reflect the latest changes.

```
| Opcode | Flags | Full Opcode | Decimal |
|:-------|:------|:------------|:-------|
| Deposit Stable | `FLAG_BOTH_TRANSFERS` | `0b11000000` | `192`
| Redeem Stable | `FLAG_BOTH_TRANSFERS` | `0b11000001` | `193` |
| Repay Stable | `FLAG_INCOMING_TRANSFER` | `0b10000000` | `128`
| Lock Collateral | `FLAG_INCOMING_TRANSFER` | `0b10000001` | `1
| Unlock Collateral | `FLAG_OUTGOING_TRANSFER` | `0b01000000` |
| Borrow Stable | `FlAG_OUTGOING_TRANSFER` | `0b01000001` | `65`
| Claim Rewards | `FLAG_OUTGOING_TRANSFER` | `0b01000010` | `66`
```

```
/*
struct Instruction {
  uint8 op_code; // [1 byte]
  bytes32 sender_address; // [32 bytes]
  one_of {
    // [deposit_stable] opcode = 192
    uint64 sequence; // [8 bytes]

    // [repay_stable] opcode = 128
    uint64 sequence; // [8 bytes]

    // [unlock_collateral] opcode = 64
    bytes32 collorateral_token_address; // [32 bytes]
    uint128 amount; // [16 bytes]

    // [borrow_stable] opcode = 65
    uint256 amount; // [16 bytes]

    // [claim rewards] opcode = 66
    // N/A for now
```

```
        // [redeem_stable] opcode = 193
        uint64 sequence; // [8 bytes]

        // [lock_collateral] opcode = 129
        uint64 sequence; // [8 bytes]
      }
    }
    */
```

[L-03] Slightly imprecise tax calculation

Line References

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-bAsset-contracts/packages/basset/src/tax_querier.rs#L12-L15

https://docs.anchorprotocol.com/developers-ethereum/fees#terra-blockchain-tax

Description

The tax amount is calculated as `amount * DENOM / DENOM + tax_rate` instead of `amount * tax_rate / DENOM`, which means the tax percentage isn't as precise (less tax is actually charged).

We see this in the test case:

```
    // normal tax
    assert_eq!(
      deduct_tax(&deps.as_mut().querier, Coin::new(50000000u128, "uu
      Coin {
        denom: "uusd".to_string(),
        amount: Uint128::new(49504950u128)
      }
    );
```

Given an amount of `50_000_000`, assuming a tax rate of 1%,the amount after tax would be `0.99 * 50_000_000 = 49500000`, but the calculated amount is `50_000_000 * 100 / 101 ~= 49504950`.

Recommended Mitigation Steps

Update the calculation to be

```
(coin.amount.checked_sub(coin.amount.multiply_ratio(
    Uint128::new(1) * tax_rate,
    DECIMAL_FRACTION,
)))?,
```

[L-04] Duplicate vesting schedules can be added

Line References

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/staking/src/contract.rs

Description

In the LP staking contract, it is possible to instantiate / add duplicate vesting schedules.

Proof of Concept

```
#[test]
fn test_instantiate_duplicate_schedules() {
  let mut deps = mock_dependencies(&[]);

  let msg = InstantiateMsg {
    anchor_token: "reward0000".to_string(),
    staking_token: "staking0000".to_string(),
    distribution_schedule: vec![
      (
        mock_env().block.time.seconds() + 100,
        mock_env().block.time.seconds() + 200,
        Uint128::from(1000000u128),
      ),
      (
        mock_env().block.time.seconds() + 100,
        mock_env().block.time.seconds() + 200,
        Uint128::from(1000000u128),
      ),
    ],
  };

  let info = mock_info("addr0000", &[]);
  let _res = instantiate(deps.as_mut(), mock_env(), info, msg).ι
}
```

Recommended Mitigation Steps

De-duplicate the new vesting schedules when it is added. It is easier done if the schedules are sorted prior.

[N-01] money-market-contracts: Market max_borrow_factor is not capped

Line References

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/money-market-contracts/contracts/market/src/contract.rs#L66

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/money-market-contracts/contracts/market/src/contract.rs#L321-L323

Description

There is no check to ensure that `max_borrow_factor` is less than 100% (`Decimal::One()`). It is therefore possible to set a borrow factor of > 1. However, the consequence of it is negligible because it would be the same as setting the value to 100% (can't borrow if there is no liquidity left).

Proof of Concept

Change the instantiation and update config test values to a value greater than 100%, such as `Decimal256::MAX`.

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/money-market-contracts/contracts/market/src/testing/tests.rs#L36

```
    let msg = InstantiateMsg {
      owner_addr: "owner".to_string(),
      stable_denom: "uusd".to_string(),
      aterra_code_id: 123u64,
      anc_emission_rate: Decimal256::one(),
      max_borrow_factor: Decimal256::MAX,
    };
```

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/money-market-contracts/contracts/market/src/testing/tests.rs#L215

```
    let msg = ExecuteMsg::UpdateConfig {
      owner_addr: None,
      interest_model: Some("interest2".to_string()),
      distribution_model: Some("distribution2".to_string()),
      max_borrow_factor: Some(Decimal256::percent(120)),
    };
```

Recommended Mitigation Steps

Ensure `max_borrow_factor` doesn’t exceed `Decimal256::one()`.

```
// add error in market/src/error.rs
pub enum ContractError {
  #[error("Setting greater than theoretical borrow factor")]
  MaxTheoreticalBorrowFactorExceeded {},
  ...
}

// in market/src/contract.rs

if msg.max_borrow_factor > Decimal256::one() {
  return Err(ContractError::MaxTheoreticalBorrowFactorExceede
}

// in update_config()
if let Some(max_borrow_factor) = max_borrow_factor {
  if max_borrow_factor > Decimal256::one() {
    return Err(ContractError::MaxTheoreticalBorrowFactorExcee
  }
  config.max_borrow_factor = max_borrow_factor;
}
```

[N-02] Incorrect description of `Safe Ratio` in documentation

Reference

https://docs.anchorprotocol.com/protocol/anchor-governance/modify-liquidation-parameters

Description

The docs say that

“A **low** `Safe Ratio` value allows for the fast liquidation of collaterals while incurring a high price impact for the collateral, while a **low** `Safe Ratio` value enforces liquidations with lower collateral price impact, albeit with slower collateral liquidation.”

The second statement describes a **high** safe ratio.

Recommended Mitigation Steps

Change to “while a **high** `Safe Ratio` value enforces liquidations with lower collateral price impact, albeit with slower collateral liquidation.”

[N-03] Typos / Grammar

Description

Do a CMD / CTRL + F for the following statements.

Recommended Mitigation Steps

1. `form` → `from`
    - `// load price form the oracle` → `// load price from the oracle`

    - `// load balance form the token contract` → `// load price from the token contract`
2. `WITHDARW` → `WITHDRAW`
    - `ALICE CAN ONLY WITHDARW 40 UST` → `ALICE CAN ONLY WITHDRAW 40 UST`
3. `// cannot liquidation collaterals` → `// cannot liquidate collaterals`
4. `// if the token contract is  already register we cannot change the address` → `// if the token contract is already registered we cannot change the address`
    - Note the double spacing between `is` and `already`
5. `inistantiation` → `instantiation`
    - `// cannot register the token at the inistantiation` → `// cannot register token at instantiation`

[N-04] Outstanding TODO

Line Reference

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/money-market-contracts/contracts/overseer/src/contract.rs#L395

Description

There's 1 remaining TODO that may not have been resolved.

```
// TODO: Should this become a reply? If so which SubMsg to make reply_on?
```

[Suggestion-01] Change interest rate model to jump rate model

Reference

https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/money-market-contracts/contracts/interest_model/src/contract.rs#L114-L135

https://github.com/compound-finance/compound-protocol/blob/master/contracts/JumpRateModel.sol

https://docs.benqi.fi/benqi-liquidity-market/protocol-parameters

Description

The current interest rate model used is a simple model that scales linearly with market utilization. A better interest rate model that Compound and its forks are using is the jump rate model, because it is more efficient at incentivizing liquidity at higher utilization rates.

Recommended Mitigation Steps

Change the interest rate model to the jump rate model, where a kink is introduced to increase the interest multiplier after a specified market utilization rate.

Alex the Entreprenerd (triage) commented:

> This looks like the most complete report.

Gas Optimizations

For this contest, 8 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **csanuragjain** received the top score from the judge.

The following wardens also submitted reports: gzeon, WatchPug, hickuphh3, 0v3rf10w, defsec, robee, *and* IllIIII.

[G-01] anchor_basset_hub contract :: execute_register_validator function

Function: execute_register_validator

Contract: https://github.com/Anchor-Protocol/anchor-bAsset-contracts/blob/master/contracts/anchor_basset_hub/src/config.rs#L114

Problem: If a Validator is already registered there is no need of further processing. Check for same is missing

Recommendation: Add a check to verify if the given validator is already whitelisted in which case directly return. Use is_valid_validator in state.rs for this purpose

[G-02] gov contract :: withdraw_voting_tokens function

Function: withdraw_voting_tokens

Contract: https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/gov/src/staking.rs#L87

Problem: Gas is wasted if withdraw_share is computed as 0

Recommendation: Add a check for withdraw_share>0, otherwise return

[G-03] gov contract :: create_poll function

Function: create_poll

Contract: https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/gov/src/contract.rs#L281

Recommendation: Change `state.poll_count += 1;` to `state.poll_count = poll_id;` to perform gas saving

[G-04] gov contract :: cast_vote

Function: cast_vote

Contract: https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/gov/src/contract.rs#L582

Problem: If amount is 0 then user vote gets wasted and also cause gas wastage

Recommendation: check `amount!=0`

[G-05] community contract :: execute function

Function: execute

Contract: https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/community/src/contract.rs#L35

Recommendation: Since both function require governance, governance check can be placed in execute instead of placing individually in UpdateConfig and Spend as done in https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-token-contracts/contracts/vesting/src/contract.rs

[G-06] anchor_basset_reward :: execute_increase_balance/execute_decrease_balance

Function: execute_increase_balance/execute_decrease_balance

Contract:
https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-bAsset-contracts/contracts/anchor_basset_reward/src/user.rs#L80
https://github.com/code-423n4/2022-02-anchor/blob/main/contracts/anchor-bAsset-contracts/contracts/anchor_basset_reward/src/user.rs#L125

Recommendation:
In both the function add a check for `amount!=0`

Alex the Entreprenerd (triage) commented:

> Probably most interesting report.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.