Starknet Part 2 Audit Report

Prepared by CodeHawks

Version 2.0

June 14, 2025

Contents

1 About CodeHawks

CodeHawks helps protect some of the world's largest protocols. It offers private and public smart contract auditing competitions, supported by a global community of industry-leading security researchers. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at codehawks.cyfrin.io.

2 Disclaimer

The CodeHawks team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit contest does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Audit Scope

```
workspace
apps
    staking
        contracts
            src
                attestation
                   attestation.cairo
                   errors.cairo
                   interface.cairo
                constants.cairo
                errors.cairo
                pool
                   eic.cairo
                   errors.cairo
                   interface.cairo
                   objects.cairo
                   pool.cairo
                   pool_member_balance_trace
                      trace.cairo
                reward_supplier
                   errors.cairo
                   interface.cairo
                   reward_supplier.cairo
                staking
                   eic.cairo
                   errors.cairo
                   interface.cairo
```

```
    objects.cairo
    staker_balance_trace
       trace.cairo
    staking.cairo
types.cairo
utils.cairo
```

5 Executive Summary

Over the course of Apr 3rd, 2025 - Apr 24th, 2025, CodeHawks auditors conducted an audit on the Starknet Part 2 smart contracts provided by Starknet. In this period, a total of 9 issues were found.

Summary

Project Name	Starknet Part 2
Repository	2025-04-starknet-part-2
Audit Timeline	Apr 3rd, 2025 - Apr 24th, 2025
Methods	Competitive Audit

Issues Found

High Risk	1
Medium Risk	1
Low Risk	7
Total Issues	9

6 Findings

6.1 High Risk Findings

6.1.1 Malicious validators can easily predict target blocks and attest to them without having to run a full node

Submitted by rex_zee, maze, bareli, hamadiftikhar, trtrth, sempermutans, umar419, iamandreiski, 0xblackadam, 0xlookman, vijayreddy, bladesec. Selected submission by: iamandreiski.

Summary The current implementation of how the target block is calculated isn't random, but rather it's pseudo-random.

This can be exploited by malicious validators under certain conditions in order to always "predict" the pseudo-random target block that they will be assigned, and use the `get_block_hash` system call of the block-in-question in order to get the hash and pass it in the `attest` function.

Vulnerability Details Considering the current implementation of how the target block is calculated:

```
fn _calculate_target_attestation_block(
            self: @ContractState,
```

```
        staking_attestation_info: StakingAttestationInfo,
        attestation_window: u16,
    ) -> u64 {
        // Compute staker hash for the attestation.
        let hash = PoseidonTrait::new()
            .update(staking_attestation_info.stake().into())
            .update(staking_attestation_info.epoch_id().into())
            .update(staking_attestation_info.staker_address().into())
            .finalize();
        // Calculate staker's block number in this epoch.
        let block_offset: u256 = hash
            .into() % (staking_attestation_info.epoch_len() - attestation_window.into())
            .into();
        // Calculate actual block number for attestation.
        let target_attestation_block = staking_attestation_info.current_epoch_starting_block()
            + block_offset.try_into().unwrap();
        target_attestation_block
    }
```

It can be easily predicted which block do the validators need to attest to.

Consider the following situation:

- Bob is a malicious validator who doesn't have a pool, but only utilizes their own stake (this is to reduce the possibility of a varying stake amount, although this can be predicted as well);
- Since epoch_id is an incrementally increasing value, this is an easily predictable value in each epoch;
- The staker address remains the same;
- The operator with which the modulo operation is performed is also a "constant" unless the admin team changes the epoch length or the attestation_window (although this is also easily predictable as well);
- Bob uses a contract which they control in order to calculate the target block that he needs to attest to once the block starts, and knowing which block that is, at the 11th block calls the Cairo syscall `get_block_hash` (if their block is within the first 10 of that epoch, or target block + 10 blocks for any other) in order to receive the block hash.
- Once they get the block_hash in-question, they can pass it in the `attest` function successfully performing the attestation and receiving rewards for it.

Bob doesn't need to run a full node or use any computation resources in order to receive the reward, successfully circumventing the attestation validation functionality.

Impact Malicious validators don't need to run a full node, or any node for that matter in order to calculate the target block that they need to attest to, as well as get the block hash further on. This has an impact of the security of the network since validation participation isn't enforced effectively and can be circumvented.

Tools Used Manual Review

Recommendations No simple recommendation exists to solve the above-mentioned problem, but there are ways to minimize it:

- Introduce true randomness via an external oracle OR some kind of an offchain system which will compute a different salt each time as part of the hash, making the target block less predictable in each epoch;
- Use a short attestation window so that validators would receive the assigned target block prior to needing to attest and have to do so within a short timeframe, further minimizing the risk for any malicious behavior.

- A third more “extreme” option would be utilizing an off-chain monitoring system for validator activity so that certain validators can be blacklisted and kicked off from the system, as currently there’s no way to remove a validator which misbehaves.

6.2 Medium Risk Findings

6.2.1 Reward is unfairly distributed to stakers because staking power is based on next epoch balance

Submitted by t0x1c, 0xsimao, trtrth, glightspeed2, iamandreiski, VulnSeekers, said, prapandey031, bladesec. Selected submission by: glightspeed2.

Summary V1 calculates total rewards based on `get_total_stake`, which returns total balance in next epoch. However, next epoch balance is "unstable", meaning it can be changed quite frequently in a single epoch.

As a result, stakers attesting in different blocks might face different rewards based on how next epoch balance was changed in the period.

Vulnerability Details

Root Cause Analysis V1 rewards model works like the following:

- For an epoch, total reward is calculated by `total_supply` and `total_stake`:

$ \text{total_reward} = \dfrac {\sqrt{ {\text{total_supply}} \centerdot \text{total_stake} } \centerdot \text{num_c}} {\text{epochs_in_year}}$

- Here, `total_stake` means protocol's total staked balance in the next epoch
  - Note: more precisely, it is total balance in the *latest* epoch. But since any balance change during the epoch will be inserted in the next epoch, latest epoch will most likely be equal to the next epoch
- This `total_reward` is accumulated to staker's `unclaimed_rewards` and pool's rewards based on the *current epoch* balance *pro rata*.

The problem is `total_reward` is dependant on next epoch's balance. Next epoch balance can be changed frequently during a single epoch, because any balance change (including stake, increase_stake, enter_delegation_pool, exit_delegation_pool etc) will be reflected to the next epoch.

Moreover, stakers are assigned different target attestation block and window. So if next epoch balance has been changed in those periods, their reward distribution will be affected by those changes.

This can lead to unfair reward distribution to stakers due to the following reasons:

- Stakers are assigned different target attestation block and they must perform attest in given attestation window. If the protocol's next epoch total balance moved in a negative way during `[current_epoch_start_block, attestation_start_block]`, they cannot do anything about them and but face reward loss
- On the contrary, protocol's balance at current epoch remained the same during that period. And reward is distributed based on current epoch balance pro rata. So it's unfair to calculate rewards based on next epoch's balance

POC Scenario

- We have three stakers Alice, Bob and Eve
- Alice and Bob stakes the same amount to the protocol
  - Alice's attestation block offset is 110
  - Bob's attestation block offset is 272
- In Alice's attestation window, the following happens
  - Eve stakes some amount to the protocol

    - Alice performs attest
    - Eve unstakes from the protocol
- Bob attests in their attestation window
- Alice gets more reward than Bob, even though they have staked the same amount to the protocol and Eve virtually didn't make any contribution to the protocol

How to Run

Apply the following diff and run `snforge test basic_stake_flow_test`:

```
diff --git a/workspace/apps/staking/contracts/src/flow_test/flows.cairo
↪  b/workspace/apps/staking/contracts/src/flow_test/flows.cairo
index b3f6f09..205ea50 100644
--- a/workspace/apps/staking/contracts/src/flow_test/flows.cairo
+++ b/workspace/apps/staking/contracts/src/flow_test/flows.cairo
@@ -17,8 +17,12 @@ use staking::test_utils::constants::UPGRADE_GOVERNOR;
 use staking::test_utils::{
     calculate_pool_member_rewards, calculate_pool_rewards, calculate_pool_rewards_with_pool_balance,
     declare_pool_contract, declare_pool_eic_contract, deserialize_option, pool_update_rewards,
-    staker_update_old_rewards,
+    staker_update_old_rewards, calculate_block_offset
 };
+use snforge_std::{start_cheat_block_number_global};
+use starknet::{get_block_number};
+use staking::staking::objects::{EpochInfo, EpochInfoTrait, InternalStakerInfoLatestTrait};
+use staking::constants::MIN_ATTESTATION_WINDOW;
 use staking::types::{Amount, Commission, Index};
 use staking::utils::{compute_rewards_per_strk, compute_rewards_rounded_down};
 use starknet::{ClassHash, ContractAddress, Store};
@@ -55,52 +59,55 @@ pub(crate) impl BasicStakeFlowImpl<
     fn test(self: BasicStakeFlow, ref system: SystemState<TTokenState>, system_type: SystemType) {
         let min_stake = system.staking.get_min_stake();
         let stake_amount = min_stake * 2;
-        let initial_reward_supplier_balance = system
-            .token
-            .balance_of(account: system.reward_supplier.address);
-        let staker = system.new_staker(amount: stake_amount * 2);
-        system.stake(:staker, amount: stake_amount, pool_enabled: true, commission: 200);
-        system.advance_epoch_and_attest(:staker);
-
-        system.increase_stake(:staker, amount: stake_amount / 2);
-        system.advance_epoch_and_attest(:staker);
+        let commission = 0;
+        let staker_alice = system.new_staker(amount: stake_amount);
+        let staker_bob = system.new_staker(amount: stake_amount);
+        let staker_eve = system.new_staker(amount: stake_amount);
+        system.stake(staker: staker_alice, amount: stake_amount, pool_enabled: false, :commission);
+        system.stake(staker: staker_bob, amount: stake_amount, pool_enabled: false, :commission);
+        system.advance_epoch();

-        let pool = system.staking.get_pool(:staker);
-        let delegator = system.new_delegator(amount: stake_amount);
-        system.delegate(:delegator, :pool, amount: stake_amount / 2);
-        system.advance_epoch_and_attest(:staker);
+        let alice_block_offset = calculate_block_offset(
+            stake: stake_amount,
+            epoch_id: system.staking.get_epoch_info().current_epoch().into(),
+            staker_address: staker_alice.staker.address,
+            epoch_len: system.staking.get_epoch_info().epoch_len_in_blocks().into(),
+            attestation_window: MIN_ATTESTATION_WINDOW
```

```
+        );
+        let bob_block_offset = calculate_block_offset(
+            stake: stake_amount,
+            epoch_id: system.staking.get_epoch_info().current_epoch().into(),
+            staker_address: staker_bob.staker.address,
+            epoch_len: system.staking.get_epoch_info().epoch_len_in_blocks().into(),
+            attestation_window: MIN_ATTESTATION_WINDOW
+        );

-        system.increase_stake(:staker, amount: stake_amount / 4);
-        system.advance_epoch_and_attest(:staker);
+        println!("alice_block_offset {:?}", alice_block_offset);
+        println!("bob_block_offset {:?}", bob_block_offset);
+        assert!(alice_block_offset < bob_block_offset, "alice_block_offset < bob_block_offset");

-        system.increase_delegate(:delegator, :pool, amount: stake_amount / 4);
-        system.advance_epoch_and_attest(:staker);
+        let initial_block = get_block_number();

-        system.delegator_exit_intent(:delegator, :pool, amount: stake_amount * 3 / 4);
-        system.advance_epoch_and_attest(:staker);
+        start_cheat_block_number_global(block_number: initial_block + alice_block_offset +
↪  MIN_ATTESTATION_WINDOW.into());
+        system.stake(staker: staker_eve, amount: stake_amount, pool_enabled: false, :commission);
+        system.attest(staker: staker_alice);
+        system.staker_exit_intent(staker: staker_eve);
+        start_cheat_block_number_global(block_number: initial_block + bob_block_offset +
↪  MIN_ATTESTATION_WINDOW.into());
+        system.attest(staker: staker_bob);

-        system.staker_exit_intent(:staker);
+        system.advance_epoch();
+        system.staker_exit_intent(staker: staker_alice);
+        system.staker_exit_intent(staker: staker_bob);
         system.advance_time(time: system.staking.get_exit_wait_window());
-
-        system.delegator_exit_action(:delegator, :pool);
-        system.delegator_claim_rewards(:delegator, :pool);
-        system.staker_exit_action(:staker);
-
-        assert!(system.token.balance_of(account: system.staking.address).is_zero());
-        assert!(system.token.balance_of(account: pool) < 100);
-        assert!(system.token.balance_of(account: staker.staker.address) == stake_amount * 2);
-        assert!(system.token.balance_of(account: delegator.delegator.address) == stake_amount);
-        assert!(system.token.balance_of(account: staker.reward.address).is_non_zero());
-        assert!(system.token.balance_of(account: delegator.reward.address).is_non_zero());
-        assert!(wide_abs_diff(system.reward_supplier.get_unclaimed_rewards(), STRK_IN_FRIS) < 100);
-        assert!(
-            initial_reward_supplier_balance == system
-                .token
-                .balance_of(account: system.reward_supplier.address)
-                + system.token.balance_of(account: staker.reward.address)
-                + system.token.balance_of(account: delegator.reward.address)
-                + system.token.balance_of(account: pool),
-        );
+        system.advance_epoch();
+        system.staker_exit_action(staker: staker_alice);
+        system.staker_exit_action(staker: staker_bob);
+
+        let alice_reward = system.token.balance_of(account: staker_alice.reward.address);
+        let bob_reward = system.token.balance_of(account: staker_bob.reward.address);
+        println!("alice reward {:?}", alice_reward);
```

```
+        println!("bob reward {:?}", bob_reward);
+        assert!(alice_reward > bob_reward, "alice_reward > bob_reward");
     }
 }
```

Console output:

```
Collected 1 test(s) from staking package
Running 1 test(s) from tests/
alice_block_offset 110
bob_block_offset 272
alice reward 79089077971181611576
bob reward 64575961752196120949
```

Impact The POC demonstrates the case where Alice receives unfair additional reward due to Eve's "flash staking".

However, an opposite scenario can happen. For example, if Eve declares unstake intent just after Alice's attestation window, Bob will receive less reward than Alice.

Overall, reward amount fluctuates as next epoch's balance changes. This is unfair for stakers because their reward changes quite randomly (depending on next epoch balance change and attestation window).

Tools Used Manual Review, Snforge

Recommendations V1 rewards model should be changed to use current epoch balance, instead of next epoch balance

```
diff --git a/workspace/apps/staking/contracts/src/staking/interface.cairo
↪  b/workspace/apps/staking/contracts/src/staking/interface.cairo
index 9495e18..3379926 100644
--- a/workspace/apps/staking/contracts/src/staking/interface.cairo
+++ b/workspace/apps/staking/contracts/src/staking/interface.cairo
@@ -38,6 +38,7 @@ pub trait IStaking<TContractState> {
     ) -> CommissionCommitment;
     fn contract_parameters_v1(self: @TContractState) -> StakingContractInfoV1;
     fn get_total_stake(self: @TContractState) -> Amount;
+    fn get_next_epoch_total_stake(self: @TContractState) -> Amount;
     fn get_current_total_staking_power(self: @TContractState) -> Amount;
     fn get_pool_exit_intent(
         self: @TContractState, undelegate_intent_key: UndelegateIntentKey,
```

```
diff --git a/workspace/apps/staking/contracts/src/staking/staking.cairo
↪  b/workspace/apps/staking/contracts/src/staking/staking.cairo
index bcfaf8e..c1a8e0a 100644
--- a/workspace/apps/staking/contracts/src/staking/staking.cairo
+++ b/workspace/apps/staking/contracts/src/staking/staking.cairo
@@ -516,6 +516,10 @@ pub mod Staking {
         }

         fn get_total_stake(self: @ContractState) -> Amount {
+            self.get_current_total_staking_power()
+        }
+
+        fn get_next_epoch_total_stake(self: @ContractState) -> Amount {
             let total_stake_trace = self.total_stake_trace;
             // Trace is initialized with a zero stake at the first valid epoch, so it is safe to
             // unwrap.
```

```
@@ -1325,11 +1329,11 @@ pub mod Staking {
         }

         fn add_to_total_stake(ref self: ContractState, amount: Amount) {
-            self.update_total_stake(new_total_stake: self.get_total_stake() + amount);
+            self.update_total_stake(new_total_stake: self.get_next_epoch_total_stake() + amount);
         }

         fn remove_from_total_stake(ref self: ContractState, amount: Amount) {
-            self.update_total_stake(new_total_stake: self.get_total_stake() - amount);
+            self.update_total_stake(new_total_stake: self.get_next_epoch_total_stake() - amount);
         }

         fn update_total_stake(ref self: ContractState, new_total_stake: Amount) {
```

6.3 Low Risk Findings

6.3.1 Updating epoch info can impact commission commitment expiration date

Submitted by glightspeed2, trtrth, ReentrantDAO, 0xadrii, ChainDefenders, bladesec. Selected submission by: trtrth.

Summary The current commission commitment mechanism is based on the Epoch mechanism, such that the commitment expiration is in epoch scale. This can cause the commitment expiration to be unexpectedly adjusted when epoch info is updated.

Vulnerability Details The function `is_commission_commitment_active()` is used to check if the commitment is active by checking current epoch with the commitment's expiration epoch.

```
    fn is_commission_commitment_active(
        self: @ContractState, commission_commitment: CommissionCommitment,
    ) -> bool {
        self.get_current_epoch() < commission_commitment.expiration_epoch
    }
```

In the mean time, the function `set_commission_commitment()` is used by the staker to set commission commitment with expiration date on the epoch `expiration_epoch()`.

```
        fn set_commission_commitment(
            ref self: ContractState, max_commission: Commission, expiration_epoch: Epoch,
        ) {
            self.general_prerequisites();
            let staker_address = get_caller_address();
            let mut staker_info = self.internal_staker_info(:staker_address);
            assert!(staker_info.unstake_time.is_none(), "{}", Error::UNSTAKE_IN_PROGRESS);
            let pool_info = staker_info.get_pool_info();
            let current_epoch = self.get_current_epoch();
            if let Option::Some(commission_commitment) = staker_info.commission_commitment {
                assert!(
                    !self.is_commission_commitment_active(:commission_commitment),
                    "{}",
                    Error::COMMISSION_COMMITMENT_EXISTS,
                );
            }
            assert!(pool_info.commission <= max_commission, "{}", Error::MAX_COMMISSION_TOO_LOW);
            assert!(expiration_epoch > current_epoch, "{}", Error::EXPIRATION_EPOCH_TOO_EARLY);
            assert!(
                expiration_epoch - current_epoch <= self.get_epoch_info().epochs_in_year(),
```

```
                "{}",
                Error::EXPIRATION_EPOCH_TOO_FAR,
            );
            let commission_commitment = CommissionCommitment { max_commission, expiration_epoch };
            staker_info.commission_commitment = Option::Some(commission_commitment);
            self.write_staker_info(:staker_address, :staker_info);
            self
                .emit(
                    Events::CommissionCommitmentSet {
                        staker_address, max_commission, expiration_epoch,
                    },
                );
        }
```

Here exists an issue that if the `epoch_info` is updated, such that epoch length/duration is updated then the expected commitment expiration date is adjusted, as a result of epoch info update. The disadvantage can be the commitment may be unexpectedly lengthened, such that it may refer to more than 1 year, which can be contradict from proposal about Commission increase

POC Add the test below to file `workspace/apps/staking/contracts/src/staking/test.cairo`

```
fn test_epoch_affect_commitment() {
    let cfg: StakingInitConfig = Default::default();
    let token_address = deploy_mock_erc20_contract(
        initial_supply: cfg.test_info.initial_supply, owner_address: cfg.test_info.owner_address,
    );
    let staking_contract = deploy_staking_contract(:token_address, :cfg);
    let staking_dispatcher = IStakingDispatcher { contract_address: staking_contract };
    stake_with_pool_enabled(:cfg, :token_address, :staking_contract);

    advance_epoch_global();

    // Set commitment.
    let staker_address = cfg.test_info.staker_address;
    let staker_info = staking_dispatcher.staker_info_v1(:staker_address);
    let max_commission = staker_info.get_pool_info().commission + 2;

    // expire in 1 year
    let expiration_epoch = staking_dispatcher.get_epoch_info().epochs_in_year();

    cheat_caller_address_once(contract_address: staking_contract, caller_address: staker_address);
    staking_dispatcher.set_commission_commitment(:max_commission, :expiration_epoch);

    // update epoch info
    // increase epoch duration
    let new_epoch_len = EPOCH_LENGTH.into() / 2;
    let new_epoch_duration = EPOCH_DURATION * 2;

    cheat_caller_address_once(
        contract_address: staking_contract, caller_address: cfg.test_info.token_admin,
    );
    let staking_config_dispatcher = IStakingConfigDispatcher { contract_address: staking_contract };
    staking_config_dispatcher
        .set_epoch_info(epoch_duration: new_epoch_duration, epoch_length: new_epoch_len);

    // advance to the expiration block/date
    let expiration_block = expiration_epoch * EPOCH_LENGTH.into();
    advance_block_number_global(expiration_block);

    // Update commission
    // revert
```

```
    cheat_caller_address_once(contract_address: staking_contract, caller_address: staker_address);
    staking_dispatcher.set_commission_commitment(max_commission + 1, expiration_epoch + 100);
}
```

Run the test and console shows:

```
Collected 1 test(s) from staking package
Running 1 test(s) from tests/
[FAIL] staking::staking::test::test_epoch_affect_commitment

Failure data:
    "Expiration epoch is too early, should be later then current epoch"

note: run with `SNFORGE_BACKTRACE=1` environment variable to display a backtrace
Tests: 0 passed, 1 failed, 0 skipped, 0 ignored, other filtered out

Failures:
    staking::staking::test::test_epoch_affect_commitment
```

It means that the expiration is not as expected.

Impact

- Unexpected adjustment of commission commitment expiration date

Tools Used Manual

Recommendations

- Consider use block number or block timestamp for expiration date

6.3.2 Delegation rewards silently lost when pool balance is less than 1e18 STRK

Submitted by hamadiftikhar, 0xsimao, hoossayn, i_atiq, calc1f4r, VulnSeekers, 0xblackadam, said. Selected submission by: VulnSeekers.

Summary The current implementation of the reward distribution logic in the staking system includes a flawed condition that leads to the silent loss (locking) of delegation pool rewards under specific scenarios, when the pool balance is less than 1e18 (STRK_IN_FRIS), the reward-per-STRK value calculated is 0, yet rewards are still transferred to the pool. These transferred rewards are not claimable by users, resulting in funds being effectively stuck.

Vulnerability Details The issue stems from the `compute_rewards_per_strk` function, defined as:

```
pub(crate) fn compute_rewards_per_strk(staking_rewards: Amount, total_stake: Amount) -> Index {
    if total_stake < STRK_IN_FRIS {
        return Zero::zero();
    }
    mul_wide_and_div(lhs: staking_rewards, rhs: BASE_VALUE, div: total_stake)
        .expect_with_err(err: StakingError::REWARDS_COMPUTATION_OVERFLOW)
}
```

This function returns `0` whenever the pool's `total_stake` is less than `STRK_IN_FRIS` (i.e., `1e18`), preventing any meaningful distribution of rewards. However, the upstream logic in the staking contract continues to forward `pool_rewards` to the pool even in this scenario:

```
fn update_pool_rewards(
    ref self: ContractState,
    staker_address: ContractAddress,
    staker_info: InternalStakerInfoLatest,
    pool_rewards: Amount,
) {
    if let Option::Some(pool_info) = staker_info.pool_info {
        let pool_contract = pool_info.pool_contract;
        let pool_dispatcher = IPoolDispatcher { contract_address: pool_contract };
        let pool_balance = self.get_pool_balance_curr_epoch(:staker_address);
        pool_dispatcher
            .update_rewards_from_staking_contract(rewards: pool_rewards, :pool_balance);
        self
            .send_rewards_to_delegation_pool(
                :staker_address,
                pool_address: pool_contract,
                amount: pool_rewards,
                token_dispatcher: self.token_dispatcher.read(),
            );
    }
}
```

Since `update_rewards_from_staking_contract` internally uses `compute_rewards_per_strk`, if `pool_balance` is less than `1e18`, the `cumulative_rewards_trace` is updated with no increase in the sigma value (since the `reward-per-STRK` is `zero`). This leads to a mismatch: rewards are transferred into the pool, but not accounted for in a way that allows pool members to claim them.

This situation creates an accumulation of unclaimable rewards in the pool, with the issue persisting until the pool balance reaches or exceeds `1e18`.

Impact

- Delegation pools with a balance less than 1e18 (STRK) receive rewards that cannot be distributed to members.
- These rewards are effectively locked and cannot be claimed, leading to a loss of protocol funds from the user's perspective.
- The issue silently drains the system of value over time as more rewards are sent without being distributable.

Tools Used

- Manual code inspection
- Logic tracing and review of:
  - update_pool_rewards
  - update_rewards_from_staking_contract
  - compute_rewards_per_strk

Recommendations While the compute_rewards_per_strk returns zero in order to prevent an overflow, channeling these funds to where it is claimable on knowing a pool balance is less than 1e18, to avoid funds getting stuck forever. Perhaps sum it up with staker's claimable rewards.

6.3.3 Missing Reward Rounding Residual Tracking

Submitted by nomadic_bear, loptus, hamadiftikhar, ngochungp295, ahmedaghadi, ibukunola. Selected submission by: ibukunola.

Summary The Pool contract's reward calculation logic (inferred as compute_rewards_rounded_down) uses integer division, discarding small rounding remainders (dust) without tracking or redistributing them. This results in negligible reward losses for delegators, as the total distributed rewards are slightly less than the allocated amount. The economic impact is minimal, and the issue is unlikely to be noticed, but tracking residuals could enhance precision and transparency in reward distribution.

Vulnerability Details The Pool contract calculates rewards for delegators using integer division, as inferred from IStakingPoolDispatcher and the test suite (test_send_rewards_to_staker):

- The rewards_info structure tracks distributed rewards but does not store residuals.
- No mechanism accumulates or redistributes dust, as confirmed by the absence of such logic in interfaces or the test suite.

Attack Path:

- A pool processes rewards for an epoch with multiple delegators (e.g., 100 delegators with 10,000 STRK each).
- compute_rewards_rounded_down discards small remainders for each delegator (e.g., 0.001 STRK per delegator).
- The total distributed rewards are less than the allocated amount (e.g., 1,000 STRK instead of 1,001 STRK).
- Over many epochs, the accumulated dust (e.g., 100 STRK for 100 delegators over 1,000 epochs) is not distributed, slightly reducing delegator rewards.

Impact Negligible reward loss (e.g., 0.001 STRK per delegator per epoch) accumulates slowly (e.g., 100 STRK for 100 delegators over 1,000 epochs, or 0.01% of 1,000,000 STRK total rewards). The loss is unlikely to be noticed by delegators.

Tools Used Manual review of Pool.

Recommendations Document reward precision limitations in user guides to enhance transparency.

6.3.4 Gas Exhaustion DoS in `claim_rewards` due to Unbounded `PoolMemberBalanceTrace`

Submitted by bareli, daniel526, chainsentryaudits, oxdev_, 0xadrii, ahmedaghadi, okhayeelizabeth. Selected submission by: chainsentryaudits.

Summary and Impact This report details a High-severity vulnerability within the Starknet Staking v2 `Pool` contract (`pool.cairo`). The mechanism designed to track pool member balances over epochs, `PoolMemberBalanceTrace`, can be forced to grow indefinitely through repeated standard user actions. Specifically, frequent calls to `add_to_delegation_pool` or `exit_delegation_pool_intent` add individual checkpoints to this trace without any bounds or coalescing mechanism.

The critical impact occurs during reward claims. The `claim_rewards` function iterates through this potentially enormous trace to calculate rewards. If a user inadvertently creates a very long trace history for their account (e.g., through frequent small deposits), the gas cost associated with iterating through these checkpoints during a subsequent `claim_rewards` call can easily exceed Starknet's transaction gas limits.

The vulnerability stems from the lack of bounds on the growth of the `PoolMemberBalanceTrace` and the linear iteration over its entire relevant history during reward calculation. This violates the implicit invariant that core

protocol functions should have predictable gas costs and should not be susceptible to DoS through the repeated use of standard, permissionless actions. The system fails to protect the operational integrity of the reward claiming process against excessive state growth triggered by user activity.

Vulnerability Details The root cause of this vulnerability lies in the `calculate_rewards` function in the Pool contract, which uses an unbounded loop to process all checkpoints:

Every time a delegator adds to their stake using `add_to_delegation_pool`, a new checkpoint is created:

Which calls:

And finally:

Crucially, there are no limits on:

1. The number of times a delegator can add to their stake
2. The minimum amount that can be added
3. The maximum number of checkpoints that can be processed

When claiming rewards via `claim_rewards`, the function calls `calculate_rewards` which must process every checkpoint:

Exploitation Steps:

1. A delegator enters a pool with a valid initial stake
2. The delegator makes hundreds or thousands of tiny stake additions
3. Each addition creates a new checkpoint in the delegator's `pool_member_epoch_balance` trace
4. When anyone attempts to call `claim_rewards` for this delegator, the function must process all checkpoints
5. With a sufficiently large number of checkpoints, the gas required exceeds block limits
6. This makes rewards permanently unclaimable, effectively locking the funds

Real-world Scenario Simulation:

Consider a delegator who legitimately enters a pool with 100 STRK tokens. Then, due to an automated staking system, they make 1,000 tiny additions of 0.000001 STRK each. Now their trace contains 1,001 checkpoints.

When they try to claim rewards, the `calculate_rewards` function must iterate through all 1,001 checkpoints, performing gas-intensive operations for each one, including:

- Fetching checkpoint data from storage
- Computing reward values using expensive multiplication and division
- Accumulating rewards

The cumulative gas cost quickly exceeds block limits, making rewards impossible to claim. Since the contract provides no mechanism to skip or consolidate checkpoints, the rewards are permanently locked.

Tools Used

- Manual Review

Recommendations The core issue is the unbounded iteration over potentially thousands of fine-grained checkpoints. To mitigate this, the number of iterations within a single `claim_rewards` call must be bounded, or the trace itself needs management.

6.3.5 Staker unable to attest block for rewards in epoch where they initiated unstake intent

Submitted by 0xsimao, 0xtheauditor, said. Selected submission by: 0xtheauditor.

Summary When a staker initiates an unstake intent, their stake is still active in the epoch they initiated the unstake intent in and only made inactive in the next epoch. Because of this stakers should still be able to attest and receive rewards in that epoch as they still have an active stake. However currently when a staker initiates an unstake intent before attesting a block to receive rewards, an error is thrown when they try to attest a block after to receive rewards within that same epoch. This is because function `update_rewards_from_attestation_contract` doesn't take into consideration what epoch the staker initiated the unstake intent in but simply throws an error if the staker has initiated an unstake intent even though they still have an active stake in the current epoch they are trying to attest the block in.

Vulnerability Details

1. Staker initiates a unstake intent by calling function `Staking::unstake_intent`. Stakers stake is still active in the current epoch and only removed in the next epoch.
2. Attestation window reaches in current epoch, staker tries to attest for a block by calling `Attestation::attest` function but receives an error about in progress unstaking even though their stake is still active in current epoch.
3. Problematic code is found in `update_rewards_from_attestation_contract` function which doesn't take into consideration the epoch the staker initiated the unstake intent in.

```
fn update_rewards_from_attestation_contract(
    ref self: ContractState, staker_address: ContractAddress,
) {
    // ...
    let mut staker_info = self.internal_staker_info(:staker_address);
    assert!(staker_info.unstake_time.is_none(), "{}", Error::UNSTAKE_IN_PROGRESS);
    // ...
}
```

Proof Of Concept The test below proves the above vulnerability. Add the code below in file `workspace/apps/staking/contracts/src/staking/test.cairo` and run the test using command `snforge test --exact staking::staking::test::test_staker_unable_to_receive_rewards`. The test does the following:

1. Staker creates an unstake intent.
2. Confirms staker's stake is still active in current epoch.
3. Staker attempts to attest for a block to receive rewards while in the current epoch but error `"Unstake is in progress, staker is in an exit window"` is thrown.

```
#[test]
#[should_panic(expected: "Unstake is in progress, staker is in an exit window")]
fn test_staker_unable_to_receive_rewards() {
    let mut cfg: StakingInitConfig = Default::default();
    general_contract_system_deployment(ref :cfg);
    let token_address = cfg.staking_contract_info.token_address;
    let staking_contract = cfg.test_info.staking_contract;

    stake_for_testing_using_dispatcher(:cfg, :token_address, :staking_contract);

    advance_epoch_global();

    let staking_dispatcher = IStakingDispatcher { contract_address: staking_contract };

    let unstake_intent_epoch = staking_dispatcher.get_current_epoch();
    let current_epoch_total_stake = staking_dispatcher.get_current_total_staking_power();
```

```
    cheat_caller_address_once(
        contract_address: staking_contract, caller_address: cfg.test_info.staker_address,
    );
    // 1. Staker creates unstake intent.
    staking_dispatcher.unstake_intent();

    advance_block_into_attestation_window(:cfg, stake: cfg.test_info.stake_amount);

    let attestation_contract = cfg.test_info.attestation_contract;
    let attestation_dispatcher = IAttestationDispatcher { contract_address: attestation_contract };
    // 2. Confirm epoch unstake intent was created is the same as the one for attesting.
    assert!(unstake_intent_epoch == staking_dispatcher.get_current_epoch());
    // 3. Confirm staker's stake is still active in the current epoch.
    assert!(current_epoch_total_stake == staking_dispatcher.get_current_total_staking_power());
    cheat_caller_address_once(
        contract_address: attestation_contract, caller_address: cfg.staker_info.operational_address,
    );
    attestation_dispatcher.attest(block_hash: Zero::zero());
}
```

Impact

- Stakers and their delegators lose legitimate rewards for the epoch in which they still have an active stake

Tools Used Tests. Check `Proof Of Concept` section.

Recommendations When staker initiates unstake intent via function `Staking::unstake_intent`, track the epoch in which the unstake intent was initiated.

In `update_rewards_from_attestation_contract` block attestation if there's unstake time set for staker and the current epoch is greater than the epoch in which the staker initiated the unstake intent in.

6.3.6 Current commission increase safeguards are insufficient and still allow for sudden commission changes

Submitted by 0xsimao, ngochungp295, 0xd4ps, 0xadrii, 0xtheauditor, said, prapandey031. Selected submission by: 0xadrii.

Summary Although the protocol implements a commission commitment safeguard to limit sudden commission changes, the design still allows sudden commission increase in certain situations, which undermines the delegator protection intended by the commission commitment safeguard.

Vulnerability Details The current implementation includes safeguards to prevent sudden changes in commissions from stakers. This is done by incorporating the concept of commission commitments, which forces stakers to configure a maximum allowed commission and commit to it for a specific amount of time. During the commitment duration, the staker can't increase the commission more than what was configured as the max commission.

Although this implementation limits sudden changes in commissions for some time, it has a fundamental flaw: once a commitment has finished, a staker can **immediately configure a new commitment** without any kind of delay that allows delegators to react to the new `max_commission`. This gives two options for delegators:

- Trust the staker and keep delegating to them when a commitment finishes. This is not ideal, given that the main goal of the commission commitment logic is to prevent delegators from being forced to trust the staker regarding commission changes.
- Always undelegate when a commission commitment is reaching an end, which is also not ideal as it could potentially lead to a loss of rewards for the next epoch when it shouldn't be necessary to undelegate in the first place, harming the reliance of the staking system due to sudden decreases in the delegated amounts.

With the current implementation, the following scenario can arise:

1. A staker has commited for some time to a certain maximum commission value (a max commission which is reasonable for delegators).
2. When the commission commitment finishes, the staker is able to immediately call `set_commission_commitment()` setting a 100% value on the `max_commission`. As shown in the following snippet, the only requirement for the new `max_commission` is to be higher that the currently configured commission:

```
// File: staking.cairo

fn set_commission_commitment(
            ref self: ContractState, max_commission: Commission, expiration_epoch: Epoch,
        ) {
            ...

            let pool_info = staker_info.get_pool_info();
            let current_epoch = self.get_current_epoch();


            ...


            assert!(pool_info.commission <= max_commission, "{}", Error::MAX_COMMISSION_TOO_LOW);

            ...
        }
```

3. Right after the `set_commission_commitment()` call, the staker can call `update_commission()` and actually configure the commission to 100%, given that the recently changed `max_commission` value allows it:

```
// File: staking.cairo

fn update_commission(ref self: ContractState, commission: Commission) {
            // Prerequisites and asserts.
            self.general_prerequisites();
            let staker_address = get_caller_address();
            let mut staker_info = self.internal_staker_info(:staker_address);

            ...

            if let Option::Some(commission_commitment) = staker_info.commission_commitment {
                if self.is_commission_commitment_active(:commission_commitment) {

                    assert!(
                        commission <= commission_commitment.max_commission,
                        "{}",
                        GenericError::INVALID_COMMISSION_WITH_COMMITMENT,
                    );

                    assert!(
                        commission != old_commission, "{}",
                        ↪  GenericError::INVALID_SAME_COMMISSION,
                    );
                } else {
                    ...
                }
            } else {
                ...
            }
```

```
            {
                let mut pool_info = staker_info.get_pool_info();
                pool_info.commission = commission;
                staker_info.pool_info = Option::Some(pool_info);
            }

            ...
        }
```

4. Finally, the staker attests for the current epoch. Because commissions are configured to 100%, the staker is able to effectively steals all of the corresponding delegator rewards for the current epoch.

Note that because stakers aren't even required to set a commission commitment in the first place, a newly-added staker can perform this attack after some time of having a reasonable commission and somehow gaining trust from delegators. The staker can then change the max commission making its value will be reflected in the system immediately, leading to a loss of rewards for delegators.

Impact Medium. The goal of the commission commitment is to minimize trust between the delegator and the staker, while preventing the staker from triggering sudden commission changes. However, as demonstrated, it is still possible for stakers to trigger sudden commission modifications, while the only way for delegators to prevent their rewards for a given epoch from being stolen via this attack is to undelegate from the staker before a commission commitment is reaching an end.

Tools Used Manual review.

Recommendations Consider updating the commission commitment logic so that it incorporates a minimum delay before the expected new `max_commission` value is actually configured (similar to a timelock design). Incorporating a delay allows delegators to:

- Anticipate wether the new `max_commission` is actually a reasonable value, and undelegate if they consider it too high.
- Not being required to trust the staker/unwillingly undelegate when a commission commitment is reaching an end.

6.3.7 Insufficient validations for minimum stake across multiple methods

Submitted by iamandreiski.

Summary Validators are bounded by a minimum stake amount which they need to deposit in order to be eligible to be registered as stakers, and subsequently being able to attest to blocks and claim rewards.

The problem is that any subsequent increases to the minimum stake (considering that there's a method to perform such actions) aren't sufficiently validated during attestations as well as stake increases which could lead to some validators being below the minimum stake.

Vulnerability Details Whenever someone wants to become a staker and be eligible for attestations/rewards, they can do so via the Staking contract's `stake` function in which there's a validation to make sure that the argument amount is greater than the minimum stake:

```
self.assert_staker_address_not_reused(:staker_address);
            assert!(amount >= self.min_stake.read(), "{}", Error::AMOUNT_LESS_THAN_MIN_STAKE);
            assert!(commission <= COMMISSION_DENOMINATOR, "{}", Error::COMMISSION_OUT_OF_RANGE);
```

The problem is that if Governance requires to increase the minimum stake amount in the system, which is a known practice in many staking protocols and can be done via:

```
fn set_min_stake(ref self: ContractState, min_stake: Amount) {
            self.roles.only_token_admin();
            let old_min_stake = self.min_stake.read();
            self.min_stake.write(min_stake);
            self
                .emit(
                    ConfigEvents::MinimumStakeChanged { old_min_stake, new_min_stake: min_stake },
                );
        }
```

The new minimum staking amount won't be enforced anywhere for already existing stakers.

Imagine the following scenario:

- The minimum stake has been increased from 20K STRK to 40K STRK;
- Already existing validators are free to continue attesting, as the `attest` method part of the Attestation contract never enforces this limit;
- They can also continue to increase their stake via `increase_stake` but still be under the limit, as the function never validates that after the increase, the stake is still above the min_stake.

Impact If the minimum stake threshold is increased, already existing validators can continue to attest with a stake which is below the minimum threshold as well as increase already existing stake and still result in the total one being under the threshold.

Tools Used Manual Review

Recommendations Make sure that during attestations the total stake of the validator is checked against the minimum stake threshold, as well as during `increase_stake` , a similar validation is performed.