Pashov Audit Group

Starknet Security Review

July 31st 2025 - August 15th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Starknet Staking

Starknet Staking introduces the first stage of a permissionless on-chain staking protocol, implemented in Cairo, with both direct staking and stake delegation. This stage focuses on testing the protocol's economic incentive mechanisms, including a minting-curve-based reward model, staker-defined commission rates, and a withdrawal lockup.

5. Executive Summary

A time-boxed security review of the **starkware-libs/starknet-staking** and **starkware-libs/starkware-starknet-utils** repositories was done by Pashov Audit Group, during which Pashov Audit Group engaged to review **Starknet Staking**. A total of **9** issues were uncovered.

Protocol Summary	
Project Name	Starknet Staking
Protocol Type	Delegated Staking Framework
Timeline	July 31st 2025 - August 15th 2025

Review commit hashes:

- e4f75ea411d1a46abdb5127d0abcc5b3891e9300 (starkware-libs/starknet-staking)
- 142ba4fd8525368690969f623c682e33230ce866 (starkware-libs/starkware-starknet-utils)

Fixes review commit hash:

- 5c11a5689f2d2e08ffecebf30d3e49569accbfca (starkware-libs/starknet-staking)

Class-hashes of Staking contracts:

- Staking: `0x05a93367d9e4fd00d9b17575cc52f70f8b48c3926da015cc7e87a3994f1c63a7`
- RewardSupplier: `0x05b312a712b50de7059ae1ad02f4445568ecc8ca49765989175a40f404cfceb9`
- MintingCurve: `0x0160a75d5bed7570e3eaab102286940d28242fc5211bd1503da7e7414b707f52`
- Pool: `0x069565cd9bd273116d8829ba461298a04bfe09adb0c4ac470e7bb66bcae84ab5`

Scope

`constants.cairo` `errors.cairo` `minting_curve.cairo` `pool.cairo` `interface.cairo` `interface_v0.cairo` `objects.cairo` `utils.cairo` `reward_supplier.cairo` `eic_v1_v2.cairo` `trace.cairo` `replaceability.cairo` `roles.cairo` `iterable_map.cairo` `time.cairo` `identity.cairo` `staking.cairo`

6. Findings

Findings count

Severity	Amount
Medium	1
Low	8
Total findings	**9**

Summary of findings

ID	Title	Severity	Status
[M-01]	Staker may exploit delegators by setting commission pre-migration	Medium	**Resolved**
[L-01]	Yearly mint calculation discrepancy: documentation vs implementation	Low	**Resolved**
[L-02]	Incorrect contract version numbers	Low	**Resolved**
[L-03]	Uncleared `staker_pool_info` left upon unstake action	Low	Acknowledged
[L-04]	Zero-amount initial undelegation intents allows `Event` spamming	Low	Acknowledged
[L-05]	Incorrect custom error description	Low	**Resolved**
[L-06]	The `transfer_from` / `transfer` functions do not check the return value	Low	**Resolved**
[L-07]	Disabled tokens still stakable via pools	Low	Acknowledged
[L-08]	Updating `attestation_window` may cause missed rewards	Low	Acknowledged

Medium findings

[M-01] Staker may exploit delegators by setting commission pre-migration

Severity

Impact: Medium

Likelihood: Medium

Description

The `set_commission()` function allows a staker with `V1` info to set `staker_pool_info.commission` without checking that they actually have migrated to `V2`. This is because the `internal_staker_info()` invoked to fetch the `staker_info` simply returns the available `V1` info.

```
>>      let staker_info = self.internal_staker_info(:staker_address);
        assert!(staker_info.unstake_time.is_none(), "{}", Error::UNSTAKE_IN_PROGRESS);

        let staker_pool_info_mut = self.staker_pool_info.entry(staker_address);
        if let Option::Some(old_commission) = staker_pool_info_mut.commission.read() {
            //..
        } else {
>>          staker_pool_info_mut.commission.write(Option::Some(commission));
            self.emit(Events::CommissionInitialized { staker_address, commission });
        }
```

Since the `staker_pool_info` is a `V2` feature, a staker in `V1` does not have it yet, and as such, the `else` block will be executed.

After this, this staker can simply invoke `set_open_for_delegation()` function and deploy any `BTC` pool. Notice that here, he manages to pass the checks because `staker_pool_info.commission` has already been set above:

```
    let staker_info = self.internal_staker_info(:staker_address);
    // @audit commision was already set above
>>  let staker_pool_info_mut = self.staker_pool_info.entry(staker_address);
    assert!(staker_info.unstake_time.is_none(), "{}", Error::UNSTAKE_IN_PROGRESS);
    assert!(self.does_token_exist(:token_address), "{}", Error::TOKEN_NOT_EXISTS);
    assert!(
        !staker_pool_info_mut.has_pool_for_token(:token_address),
        "{}",
        Error::STAKER_ALREADY_HAS_POOL,
    );
    // @audit passes
>>  let commission = staker_pool_info_mut.commission();

    // Deploy delegation pool contract.
```

```
    let pool_contract = self
        .deploy_delegation_pool_from_staking_contract(
            :staker_address,
            staking_contract: get_contract_address(),
            :token_address,
            :commission,
        );
    // @audit Pool added
>>  staker_pool_info_mut.pools.write(pool_contract, token_address);
    // Initialize the delegated balance trace.
    self.initialize_staker_delegated_balance_trace(:staker_address, :pool_contract);
```

Upto this point, delegators can simply delegate their `BTC` tokens to this pool since the trace has been initialized with the current epoch.

Assumptions

- The staker sets a very small commission of `2%`.
- The staker already had `STRK` pool in `V1` with `pool_info._deprecated_commission` of `40%`, which he has not migrated.
- New delegators have onboarded their new `BTC` pool with knowledge that commission is `2%`.

Now the staker is about to perform `attestation` which should distribute and split rewards based on the `staker_pool_info.commission` i.e `2%` He then invokes `staker_migration()`:

```
    fn staker_migration(ref self: ContractState, staker_address: ContractAddress) {
        // @audit Assertion passes
        assert!(
>>          self.staker_own_balance_trace.entry(staker_address).is_empty(),
            "{}",
            Error::STAKER_INFO_ALREADY_UPDATED,
        );
        // Migrate staker pool info.
        let internal_staker_info = self.internal_staker_info(:staker_address);
        let staker_pool_info_mut = self.staker_pool_info.entry(staker_address);
        let mut pool_contract = Option::None;
        // @audit fetches his STRK pool from v1
>>      if let Option::Some(pool_info) = internal_staker_info._deprecated_pool_info {
            pool_contract = Option::Some(pool_info._deprecated_pool_contract);
            let token_address = STRK_TOKEN_ADDRESS;
            // @audit old commision from v1 fetched
>>          let commission = pool_info._deprecated_commission;
            // @audit previously set commission is overwritten
>>          staker_pool_info_mut.commission.write(Option::Some(commission));
            staker_pool_info_mut.pools.write(pool_contract.unwrap(), token_address);
        }
        // Note: Staker might have a commission commitment only if he has a pool.
        staker_pool_info_mut
            .commission_commitment
            .write(internal_staker_info._deprecated_commission_commitment);
        // @audit staker_own_balance_trace is then populated here
>>      self.migrate_staker_balance_trace(:staker_address, :pool_contract);
```

```
        // Add staker address to the stakers vector.
        self.stakers.push(staker_address);
    }
```

As seen, the first assertion passes because upto this point, the staker's `staker_own_balance_trace` mapping is empty. This is a `V2` feature, and since he has not performed any stake here, there is no record present.

The function then checks if he has a `STRK` pool which he does from `V1`. The `pool_info._deprecated_commission` i.e `40%` from `V1` is then retrieved and set to `staker_pool_info_mut.commission` and that pool also added to his `staker_pool_info_mut.pools`. As such, the previously set `2%` commission that the `BTC` delegators subscribed to is overwritten and updated to `40%`.

The staker then performs attestation, and the following happens in `calculate_staker_pools_rewards()` via the `update_rewards_from_attestation_contract()` call.

```
            // @audit 40% is fetched
>>          let commission = staker_pool_info.commission();
            let curr_epoch = self.get_current_epoch();
            for (pool_contract, token_address) in staker_pool_info.pools {
                if !self.is_active_token(:token_address, :curr_epoch) {
                    continue;
                }
                let pool_balance_curr_epoch = self
                    .get_staker_delegated_balance_curr_epoch(
                        :staker_address, :pool_contract, :curr_epoch,
                    );
                let (epoch_rewards, total_stake) = if token_address == STRK_TOKEN_ADDRESS {
                    (strk_epoch_rewards, strk_total_stake)
                } else {
                    (btc_epoch_rewards, btc_total_stake)
                };
                // Calculate rewards for this pool.
                let pool_rewards_including_commission = if total_stake.is_non_zero() {
                    mul_wide_and_div(
                        lhs: epoch_rewards,
                        rhs: pool_balance_curr_epoch.to_amount_18_decimals(),
                        div: total_stake,
                    )
                        .expect_with_err(err: GenericError::REWARDS_ISNT_AMOUNT_TYPE)
                } else {
                    Zero::zero()
                };
                // @audit staker's commission_rewards is calculated based on 40% even for BTC
stakes
>>              let (commission_rewards, pool_rewards) = self
                    .split_rewards_with_commission(
                        rewards_including_commission: pool_rewards_including_commission,
                        :commission,
                    );

                // @audit Staker accumulates excess
```

```
>>              total_commission_rewards += commission_rewards;
                ---snip---
            }
```

As seen, the `BTC` delegators are rugged as they receive far less `pool_rewards` than they should.

This is an issue because by normal flow, a staker is not supposed to be in a position to increase their `commission` once set if they lack `commission_commitment` as enforced in `update_commission()`.

```
            else {
>>              assert!(commission < old_commission, "{}", GenericError::INVALID_COMMISSION);
            }
```

Recommendations

Add this check in `set_commission()`:

```
        // Assert the staker has migrated.
+       assert!(
+           self.staker_own_balance_trace.entry(staker_address).is_non_empty(),
+           "{}",
+           Error::STAKER_INFO_NOT_UPDATED,
+       );
```

Low findings

[L-01] Yearly mint calculation discrepancy: documentation vs implementation

There is a discrepancy between the documented behavior and actual implementation of the yearly mint reward calculation. The comments describe a simple linear inflation mechanism, but the implementation uses a completely different square-root-based formula, leading to different economic outcomes than documented.

The Discrepancy:

1. **Comment describes linear inflation:**

```
// The numerator of the inflation rate. The denominator is C_DENOM. C_NUM / C_DENOM is the
// fraction of the total supply that can be minted in a year.
```

This suggests the formula should be:

```
yearly_mint = total_supply * (C_NUM / C_DENOM)
```

1. **Implementation uses square-root formula:**

```
fn compute_yearly_mint(
    self: @ContractState, total_stake: Amount, total_supply: Amount,
) -> Amount {
    let stake_times_supply: u256 = total_stake.wide_mul(total_supply);
    self.c_num.read().into() * stake_times_supply.sqrt() / C_DENOM.into()
}
```

Actual formula:

```
yearly_mint = C_NUM * sqrt(total_stake * total_supply) / C_DENOM
```

This discrepancy is critical as it affects the entire economic model of the protocol and must be resolved immediately to prevent confusion and potential financial miscalculations by all stakeholders.

[L-02] Incorrect contract version numbers

The `staking.cairo` contract declares `CONTRACT_VERSION: felt252 = '2.0.0'`, but this is supposed to be version `3.0.0` (the new BTC staking version) since the currently deployed contract is already version 2.0.0.

This creates confusion for version tracking when the `version()` functions are queried.

Note that there is also an inconsistency in other contracts, for example, the minting_curve contract has significant changes, but its CONTRACT_VERSION still points at `1.0.0` this should be changed. The same applies for the pool contract, as it also points at `2.0.0` but should be `3.0.0` to show the pool contract is for the new BTC pools too.

Recommendation

Update the contract version to reflect the correct versions Example;

```
// staking.cairo
pub const CONTRACT_VERSION: felt252 = '3.0.0';
```

[L-03] Uncleared `staker_pool_info` left upon unstake action

When a staker performs unstake action, their pool info remains intact, which includes their added `pools`, `commission`, and `commission_commitment`.

However, this data nolonger serves any purpose as there is no functionality that utilizes it anymore. This therefore results in orphaned data within the system.

Mitigation

Clear this mapping during `unstake_action()`:

```
    ---snip---
    // Return delegated stake to pools and zero their balances.
    self.transfer_to_pools_when_unstake(:staker_address, :staker_pool_info);

+   // now clear this mapping
+   self.staker_pool_info.write(
+       staker_address,
+       InternalStakerPoolInfoV2 {
+           commission: Option::None,
+           pools: IterableMapTrait::new(),
+           commission_commitment: Option::None,
+       }
+   );

    staker_amount
```

[L-04] Zero-amount initial undelegation intents allows `Event` spamming

The `exit_delegation_pool_intent()` function in the delegation pool contract allows a `pool member` to submit an `amount = 0` request even if they have no existing undelegate intent.

This triggers event emissions `PoolMemberExitIntent` and `PoolMemberBalanceChanged` (on the pool side) and `RemoveFromDelegationPoolIntent` and `StakeDelegatedBalanceChanged` (on staking contract) that have no real meaning.

```
    // @audit When 0, unpool_time is set to None but the rest of function flows normally
>>  if amount.is_zero() {
        pool_member_info.unpool_time = Option::None;
    } else {
        pool_member_info.unpool_time = Option::Some(unpool_time);
    }
    ...
    // pool::exit_delegation_pool_intent()
    self.emit(Events::PoolMemberExitIntent { ... });
    self.emit(Events::PoolMemberBalanceChanged { ... });

    // staking::remove_from_delegation_pool_intent()
    self.emit(Events::RemoveFromDelegationPoolIntent { ... });
    self.emit(Events::StakeDelegatedBalanceChanged { ... });
```

Because there's no restriction on repeating such `zero-amount` calls, a malicious user could repeatedly call the function to generate spam events and polluting event logs.

Recommendations

Add a check to ensure that `zero-amount` calls are only permitted if the caller already has an active undelegate intent (`unpool_time.is_some()`).

```
+   assert!(
+       !amount.is_zero() || pool_member_info.unpool_time.is_some(),
+       "{}",
+       GenericError::ZERO_AMOUNT_INITIAL_INTENT
+   );

    // @audit proceed normaly
    if amount.is_zero() {
        pool_member_info.unpool_time = Option::None;
    } else {
        pool_member_info.unpool_time = Option::Some(unpool_time);
    }
```

This ensures: - First-time exit intents must have `amount > 0` . - Zero-amount calls are only valid as cancellations of existing intents.

[L-05] Incorrect custom error description

The `describe()` functionality in `staking/contracts/src/staking/errors.cairo` aims to output a well formed string description of the custom errors defined. However, `SELF_SWITCH_NOT_ALLOWED` and `ILLEGAL_EXIT_DURATION` are not well described as done for other errors:

```
    Error::SELF_SWITCH_NOT_ALLOWED => "SELF_SWITCH_NOT_ALLOWED",
    Error::ILLEGAL_EXIT_DURATION => "ILLEGAL_EXIT_DURATION",
```

The output string therefore, is malformed.

Recommendations

```
-    Error::SELF_SWITCH_NOT_ALLOWED => "SELF_SWITCH_NOT_ALLOWED",
-    Error::ILLEGAL_EXIT_DURATION => "ILLEGAL_EXIT_DURATION",
+    Error::SELF_SWITCH_NOT_ALLOWED => "Switching in the same pool is no allowed",
+    Error::ILLEGAL_EXIT_DURATION => "Exit window is incorrect",
```

[L-06] The `transfer_from` / `transfer` functions do not check the return value

Although the `checked_transfer_from` and `checked_transfer` functions perform relatively thorough balance and allowance checks before making a transfer call, they still do not check the return value during the transfer. This may result in the transfer silently failing for other reasons, returning false, while the contract treats it as successful.

```
    fn checked_transfer_from(
        self: IERC20Dispatcher, sender: ContractAddress, recipient: ContractAddress, amount:
u256,
    ) -> bool {
        assert!(amount <= self.balance_of(account: sender), "{}",
Erc20Error::INSUFFICIENT_BALANCE);
        assert!(
            amount <= self.allowance(owner: sender, spender: get_contract_address()),
            "{}",
            Erc20Error::INSUFFICIENT_ALLOWANCE,
        );
        self.transfer_from(:sender, :recipient, :amount)
    }

    fn checked_transfer(self: IERC20Dispatcher, recipient: ContractAddress, amount: u256) ->
bool {
        assert!(
            amount <= self.balance_of(account: get_contract_address()),
            "{}",
            Erc20Error::INSUFFICIENT_BALANCE,
        );
        self.transfer(:recipient, :amount)
    }
```

It is recommended to check the return values of `transfer` and `transfer_from` function calls.

[L-07] Disabled tokens still stakable via pools

Disabled tokens can still be staked through existing delegation pools despite being disabled for reward distribution. This creates a scenario where users unknowingly stake tokens that will never earn rewards, effectively losing the opportunity cost of their capital while their funds become non-productive.

The Core Issue:

Pool staking bypasses token status checks: When users stake via pools, the `add_stake_from_pool()` function doesn't verify if the token is currently active:

```
fn add_stake_from_pool(
    ref self: ContractState, staker_address: ContractAddress, amount: Amount,
) {
    // ... prerequisite checks ...
    let token_address = self
        .staker_pool_info
        .entry(staker_address)
        .get_pool_token(:pool_contract)
        .expect_with_err(Error::CALLER_IS_NOT_POOL_CONTRACT);

    // ← NO CHECK: assert!(self.is_active_token(:token_address, curr_epoch))

    // Staking proceeds regardless of token status
    self.insert_staker_delegated_balance(:staker_address, :pool_contract, delegated_balance:
new_delegated_stake);
    self.add_to_total_stake(:token_address, amount: normalized_amount);
}
```

Disabled tokens receive zero rewards: During reward calculation, disabled tokens are completely excluded:

```
fn calculate_staker_pools_rewards(/* ... */) -> (Amount, Amount, Array<(ContractAddress,
NormalizedAmount, Amount)>) {
    for (pool_contract, token_address) in staker_pool_info.pools {
        if !self.is_active_token(:token_address, :curr_epoch) {
            continue; // ← Disabled tokens get ZERO rewards
        }
        // Only active tokens participate in reward calculation
    }
}
```

Users receive no indication of token status: Pool interfaces don't prevent staking or warn users about disabled tokens.

Recommendations

Add active token checks to pool staking:

```
fn add_stake_from_pool(
    ref self: ContractState, staker_address: ContractAddress, amount: Amount,
) {
    // ... existing checks ...
    let token_address = self
        .staker_pool_info
        .entry(staker_address)
        .get_pool_token(:pool_contract)
        .expect_with_err(Error::CALLER_IS_NOT_POOL_CONTRACT);

    // ← ADD: Prevent staking disabled tokens
    let curr_epoch = self.get_current_epoch();
    assert!(
        self.is_active_token(:token_address, :curr_epoch),
        "{}",
        Error::TOKEN_NOT_ACTIVE
    );
```

```
    // Continue with staking...
}
```

Emit warnings when tokens are disabled:

```
#[derive(Drop, starknet::Event)]
struct TokenDisabledWarning {
    token_address: ContractAddress,
    affected_pools: Span<ContractAddress>,
    users_should_unstake: bool,
}

fn disable_token(ref self: ContractState, token_address: ContractAddress) {
    // ... existing disable logic ...

    let affected_pools = self.get_pools_for_token(:token_address);
    self.emit(TokenDisabledWarning {
        token_address,
        affected_pools,
        users_should_unstake: true
    });
}
```

[L-08] Updating `attestation_window` may cause missed rewards

When the `set_attestation_window` function is called to modify `attestation_window`, the change is applied directly to the current epoch.

```
fn set_attestation_window(ref self: ContractState, attestation_window: u16) {
    self.roles.only_app_governor();
    assert!(
        attestation_window >= MIN_ATTESTATION_WINDOW, "{}", Error::ATTEST_WINDOW_TOO_SMALL,
    );
    let old_attestation_window = self.attestation_window.read();
    self.attestation_window.write(attestation_window);
    self
        .emit(
            Events::AttestationWindowChanged {
                old_attestation_window, new_attestation_window: attestation_window,
            },
        );
}
```

Since `attestation_window` is involved in calculating `target_attestation_block` and the attestation window, modifying it may result in the current block being within a staker's attestation window before the change, but after the modification, the current block has already passed that staker's attestation window. If the staker did not submit an attestation before the change, they would miss rewards for that epoch.

Recommendations

It is recommended that adjustments to `attestation_window` via `set_attestation_window` be applied in the next epoch.