Zellic

November 22, 2024

Prepared for
Lotem Kahan
StarkWare Industries

Prepared by
William Bowling
Jinseo Kim
Ulrich Myhre
Zellic

Starknet Staking

Cairo Application Security Audit

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for StarkWare Industries from September 3rd to September 16th, 2024, and from November 11th to November 15th, 2024. During this engagement, Zellic reviewed Starknet Staking's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Can a malicious user steal or lock the staked assets?
- Is the pool delegation feature correctly implemented?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Starknet Staking contracts, we discovered 15 findings. One critical issue was found. Three were of high impact, four were of medium impact, four were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of StarkWare Industries in the Discussion section (5. ↗).

Note that findings 3.1 through 3.7 were identified during the first audit period, while findings 3.8 through 3.15 were uncovered during the second audit period.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	1
🟧 High	3
🟨 Medium	4
🟩 Low	4
⬜ Informational	3

2. Introduction

2.1. About Starknet Staking

StarkWare Industries contributed the following description of Starknet Staking:

> Staking on Starknet is designed to enhance network security and decentralization by allowing users to stake their STRK tokens directly or delegate them to other validators. The architecture is modular, with different contracts handling specific responsibilities to ensure flexibility, security, and ease of upgrades. Starknet staking is a first step towards decentralisation of Starknet following Starknet SNIP 18 ↗.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Architecture risks. This encompasses potential hazards originating from the blueprint of a system, which involves its core validation mechanism and other architecturally significant constituents influencing the system's fundamental security attributes, presumptions, trust mode, and design.

Arithmetic issues. This includes but is not limited to integer overflows and underflows, floating-point associativity issues, loss of precision, and unfavorable integer rounding.

Implementation risks. This encompasses risks linked to translating a system's specification into practical code. Constructing a custom system involves developing intricate on-chain and off-chain elements while accommodating the idiosyncrasies and challenges presented by distinct programming languages, frameworks, and execution environments.

Availability. Denial-of-service attacks are another leading issue in custom systems. Issues including but not limited to unhandled panics, unbounded computations, and incorrect error handling can potentially lead to consensus failures.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (5. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Starknet Staking Contracts

Type	Cairo
Platform	Starknet

Target	starknet-staking
Repository	https://github.com/starkware-libs/starknet-staking ↗
Version	`1fb6d430593ac64fbdc886cb165dc2e73353a289`
Programs	`apps/staking/contracts/**/*.cairo` `packages/contracts/**/*.cairo`

Target	starknet-staking
Repository	https://github.com/starkware-libs/starknet-staking ↗
Version	`e4f75ea411d1a46abdb5127d0abcc5b3891e9300`

Programs	`apps/staking/contracts/src/**/*.cairo`
	`apps/staking/contracts/src/utils.cairo`
	`packages/contracts/src/components/roles/*.cairo`
	`packages/contracts/src/components/replaceability/*.cairo`
	`packages/contracts/src/types/time.cairo`
	`apps/staking/L1/starkware/solidity/stake/*.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of three person-weeks. The assessment was conducted by three consultants over the course of two calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

William Bowling
Engineer
vakzz@zellic.io

Jinseo Kim
Engineer
jinseo@zellic.io

Ulrich Myhre
Engineer
unblvr@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 3, 2024	Kick-off call
September 3, 2024	Start of primary review period
September 16, 2024	End of primary review period
November 11, 2024	Start of second primary review period
November 15, 2024	End of second primary review period

3. Detailed Findings

3.1. Authorization modifiers that do not perform any checks

Target	MintManager		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The MintManager contract uses the `onlyAllowanceGovernor` and `onlyStopGovernor` modifiers for the purpose of authorization:

```
function approve(address account, uint256 amount)
    external onlyAllowanceGovernor {
    // ...
}

function increaseAllowance(address account, uint256 amount)
    external onlyAllowanceGovernor {
    // ...
}

function decreaseAllowance(address account, uint256 amount)
    external onlyAllowanceGovernor {
    // ...
}

function stopAllowance(address account) external onlyStopGovernor {
    // ...
}
```

However, those modifiers do not perform any checks:

```
modifier onlyAllowanceGovernor() {
    _;
}

modifier onlyStopGovernor() {
    _;
}
```

Impact

A malicious user can arbitrarily mint the associated token.

Recommendations

Consider adding the necessary check to the modifiers.

Remediation

This issue has been acknowledged by StarkWare Industries, and a fix was implemented in commit `e8e2ab58` ↗.

3.2. Using the transaction-initiator address for authentication purposes

Target	Project Wide		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The Staking contract processes the staking/unstaking request, manages the list of stakers, and stores the staked assets. Because of the structure where a user invokes the Operator contract, which then invokes the Staking contract, it cannot use the `get_caller_address` function for the purpose of authentication.

The Staking contract tries to resolve this issue by using the transaction-initiator address for the purpose of authentication:

```
fn stake(
    ref self: ContractState,
    reward_address: ContractAddress,
    operational_address: ContractAddress,
    amount: u128,
    pool_enabled: bool,
    commission: u16,
) -> bool {
    // ...
    let staker_address = get_tx_info().account_contract_address;
    // ...
}
```

However, this pattern allows a malicious contract that is either directly or indirectly invoked by a staker to interact with the Staking contract as the staker.

Impact

A malicious actor can create a phishing contract that steals the assets of the user without asking for explicit permission.

Recommendations

Consider using the caller address of the Operator contract for purpose of authentication.

Remediation

This issue has been acknowledged by StarkWare Industries, and a fix was implemented in commit `2c81b5f0` ↗.

3.3. No sender check on the `on_receive` function

Target	reward_supplier.cairo		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The Reward Supplier contract has the `on_receive` function, which processes the assets received through the StarkGate protocol:

```
fn on_receive(
    ref self: ContractState,
    l2_token: ContractAddress,
    amount: u256,
    depositor: EthAddress,
    message: Span<felt252>
) -> bool {
    let amount_low:
    u128 = amount.try_into().expect_with_err(Error::AMOUNT_TOO_HIGH);
    let mut l1_pending_requested_amount =
    self.l1_pending_requested_amount.read();
    if amount_low > l1_pending_requested_amount {
        self.l1_pending_requested_amount.write(Zero::zero());
        return true;
    }
    l1_pending_requested_amount -= amount_low;
    self.l1_pending_requested_amount.write(l1_pending_requested_amount);
    true
}
```

However, because this function does not check if the caller is StarkGate protocol, anyone can invoke this function, which decreases the variable `l1_pending_requested_amount`.

Impact

A malicious user can invoke this function in order to let the Reward Supplier contract request minting tokens, even though it is not needed to do so.

Recommendations

Consider checking if the caller of the `on_receive` function is StarkGate protocol.

Remediation

This finding was brought to our attention by StarkWare Industries prior to the official report being submitted.

This issue has been acknowledged by StarkWare Industries, and a fix was implemented in commit `695e2cad` ↗.

3.4. Operational address can be overwritten by another user

Target	staking.cairo		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The Staking contract stores the operational address of the staker in two ways. The `operational_address` field of the `staker_info` struct stores the staker-to-operational address mapping, and the storage variable `operational_address_to_staker_address` stores the operational address-to-staker mapping.

The `stake` function prevents a new staker from specifying their operational address as the operational address of another staker:

```
fn stake(
    ref self: ContractState,
    reward_address: ContractAddress,
    operational_address: ContractAddress,
    amount: u128,
    pooling_enabled: bool,
    commission: u16,
) -> bool {
    // ...
    assert_with_err(
        self
            .operational_address_to_staker_address
            .read(operational_address)
            .is_zero(),
        Error::OPERATIONAL_EXISTS
    );
    // ...
}
```

However, the `change_operational_address` function, which allows the staker to change their operational address, does not have such a check:

```
fn change_operational_address(
    ref self: ContractState, operational_address: ContractAddress
) -> bool {
    self.assert_is_unpaused();
    self.roles.only_operator();
    self.update_global_index_if_needed();
    let staker_address = get_tx_info().account_contract_address;
    let mut staker_info = self.get_staker_info(:staker_address);
    self
        .operational_address_to_staker_address
        .write(staker_info.operational_address, Zero::zero());
    let old_address = staker_info.operational_address;
    staker_info.operational_address = operational_address;
    self.staker_info.write(staker_address, Option::Some(staker_info));
    self.operational_address_to_staker_address.write(operational_address,
    staker_address);
    self
        .emit(
            Events::OperationalAddressChanged {
                staker_address, new_address: operational_address, old_address
            }
        );
    true
}
```

Impact

A malicious user can change their operational address to the operational address of another staker in order to remove the operational address-to-staker mapping information of the staker from the contract.

Recommendations

Consider checking that the operational address is not taken by another user in the `change_operational_address` function.

Remediation

This issue has been acknowledged by StarkWare Industries, and a fix was implemented in commit `fb6c3a88` ↗.

3.5. Operational address can be preoccupied by another user

Target	staking.cairo		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

Since the Staking contract uses the operational address-to-staker mapping, one operational address cannot handle multiple stakers, and choosing an operational address that is already in use is prevented (although Finding 3.4. ↗ describes the case that incorrectly allows this). This means that any user can preoccupy the operational address in order to prevent the address from being the operational address of another staker.

Impact

If a user is about to stake with operational address X, an attacker who knows address X can set their operational address to X in order to prevent the user from staking with operational address X.

Recommendations

Consider validating the signature from the operational address in order to prevent a malicious user from occupying it.

Remediation

This issue has been acknowledged by StarkWare Industries. They provided the following response:

> We realized that we don't validate the staker ownership of the operational address upon initial staking. We are not planning to fix this in this version, in order to reduce complexity and integration issues at this stage. The attacker will have to pay the initial stake for each operational address they steal, which reduce the potential of the attack. The attacked staker remedy is simply to use a different address.

3.6. Possible reentrancy, assuming that the associated ERC-20 token invokes untrusted contracts

Target	Project Wide		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

Overall, much of the logic in the codebase is vulnerable to reentrancy, provided that the associated ERC-20 token passes the execution context to untrusted contracts.

For example, the `claim_rewards`, `claim_delegation_pool_rewards`, and `unstake_action` functions of the Staking contract transfer the asset outside and then change the state:

```
fn claim_rewards(ref self: ContractState, staker_address: ContractAddress) ->
    u128 {
    // ...
    let amount = staker_info.unclaimed_rewards_own;
    let erc20_dispatcher = self.erc20_dispatcher.read();
    self
        .send_rewards_to_staker(
            :staker_address, :reward_address, :amount, :erc20_dispatcher
        );
    staker_info.unclaimed_rewards_own = 0;
    // ...
}
```

```
fn claim_delegation_pool_rewards(
    ref self: ContractState, staker_address: ContractAddress
) -> u64 {
    // ...
    self
        .send_rewards_to_delegation_pool(
            :staker_address,
            :pool_address,
            amount: updated_pool_info.unclaimed_rewards,
            :erc20_dispatcher
        );
    updated_pool_info.unclaimed_rewards = 0;
    // ...
}
```

```
fn unstake_action(ref self: ContractState, staker_address: ContractAddress) ->
    u128 {
    // ...
    let staker_amount = staker_info.amount_own;
    erc20_dispatcher.transfer(recipient: staker_address, amount:
    staker_amount.into());

    self.transfer_to_pool_when_unstake(:staker_address, :staker_info);
    self.remove_staker(:staker_address, :staker_info);
    // ...
}
```

Impact

The impact of this vulnerability is mitigated because the ERC-20 token, which StarkWare Industries plans to use, does not invoke any untrusted contracts and is under the control of StarkWare Industries. Nonetheless, this finding documents the antipattern of the codebase.

Recommendations

Consider refactoring the codebase to modify the state before any external calls.

Remediation

This issue has been acknowledged by StarkWare Industries. They provided the following response:

> This staking contract is specifically bound with Starknet token, which is not untrusted. We will make an effort to fix these issue nonetheless for the fix commit, within development effort constraints.

3.7. Sole governance admin can revoke their own role

Target	roles.cairo		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `remove_governance_admin` function allows the governance admin to revoke the governance-admin permission of another account:

```
fn remove_governance_admin(
    ref self: ComponentState<TContractState>, account: ContractAddress
) {
    let event = Event::GovernanceAdminRemoved(
        GovernanceAdminRemoved {
            removed_account: account, removed_by: get_caller_address()
        }
    );
    self._revoke_role_and_emit(role: GOVERNANCE_ADMIN, :account, :event);
}
```

However, it is not checked if there is another governance admin that can continue to perform governance action.

Impact

The governance-admin role can be mistakenly renounced, which is unrecoverable.

Recommendations

Consider preventing a sole governance admin from renouncing their own role.

Remediation

This issue has been acknowledged by StarkWare Industries, and a fix was implemented in commit 005d8b68 ↗.

4. Detailed Findings II

4.1. Incorrect reward transfers when switching to exiting pool

Target	staking.cairo		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

If a delegator decides to switch pools, they call `switch_delegation_pool` on the current pool, which will end up calling `switch_staking_delegation_pool` in the staking contract. Amongst other checks, this will cause the rewards for the new pool's staker to be updated before `enter_delegation_pool_from_staking_contract` is called on the new pool.

If the user already has delegated to the new pool, this will cause their rewards to be updated and their delegated amount to increase.

```
/// This function is called by the staking contract to enter the pool during a
    pool switch.
fn enter_delegation_pool_from_staking_contract(
    ref self: ContractState, amount: Amount, index: Index, data:
    Span<felt252>
) {
    // [SNIP]

    // Create or update the pool member info, depending on whether the pool
    member exists,
    // and then commit to storage.
    let pool_member_info = match self.pool_member_info.read(pool_member) {
        Option::Some(mut pool_member_info) => {
            // Pool member already exists. Need to update pool_member_info to
    account for
            // the accrued rewards and then update the delegated amount.
            assert_with_err(
                pool_member_info.reward_address
    == switch_pool_data.reward_address,
                Error::REWARD_ADDRESS_MISMATCH
            );
            self.update_rewards(ref :pool_member_info, updated_index: index);
            pool_member_info.amount += amount;
            pool_member_info
        },
    // [SNIP]
```

```
fn update_rewards(
    self: @ContractState, ref pool_member_info: InternalPoolMemberInfo,
    updated_index: Index
) {
    let interest: Index = updated_index - pool_member_info.index;
    pool_member_info.index = updated_index;
    let rewards_including_commission = compute_rewards_rounded_down(
        amount: pool_member_info.amount, :interest
    );
    let commission_amount = compute_commission_amount_rounded_up(
        :rewards_including_commission, commission: pool_member_info.commission
    );
    let rewards = rewards_including_commission - commission_amount;
    pool_member_info.unclaimed_rewards += rewards;
    pool_member_info.commission = self.commission.read();
}
```

The issue is that if a delegator switches to pool where they already have a partial exit happening, then their `unclaimed_rewards` will increase, but no more rewards are transferred to the pool to cover it. When the delegator completes their exit, the pool will try to transfer all of their `unclaimed_rewards`, which can fail as the pool may not have enough tokens to cover the transfer.

Impact

When this occurs, the pool does not have the required token balance to cover all the rewards owed to its delegates.

Recommendations

When the staking contract updates the rewards for a pool, those tokens should be transferred to the pool so that they may be distributed when required.

Remediation

This issue has been acknowledged by StarkWare Industries, and a fix was implemented in commit `eeba818e` ↗.

4.2. Rewards stuck in pool on commission change

Target	staking.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

A staker with a pool has the ability to set a commission, allowing them to take a percentage of any delegator's rewards. They also have the ability to adjust the commission rate, provided it remains lower than the previous rate.

The pool contract also keeps track of the commission for each pool member, which is updated each time after the pool members' rewards are calculated:

```
fn update_rewards(
    self: @ContractState, ref pool_member_info: InternalPoolMemberInfo,
    updated_index: Index
) {
    let interest: Index = updated_index - pool_member_info.index;
    pool_member_info.index = updated_index;
    let rewards_including_commission = compute_rewards_rounded_down(
        amount: pool_member_info.amount, :interest
    );
    let commission_amount = compute_commission_amount_rounded_up(
        :rewards_including_commission, commission: pool_member_info.commission
    );
    let rewards = rewards_including_commission - commission_amount;
    pool_member_info.unclaimed_rewards += rewards;
    pool_member_info.commission = self.commission.read();
}
```

The issue is that after the commission is changed, the `staking.update_rewards` function will be using the new commission (causing more rewards to go to the pool), but the `pool.update_rewards` function will be using the previously saved `pool_member_info.commission`. Since this only gets updated when the delegate performs an action on the pool, any extra rewards accrued during that time from the changed commission will be stuck on the pool.

Impact

When the commission for a pool is changed, any extra rewards that should be going to the delegate will end up being stuck in the pool with no way to claim them by either the delegates or the staker. The longer a delegate waits to claim their rewards, the more rewards they stand to lose.

Recommendations

Allowing the commission rate to be changed requires keeping track of it for each pool member and complicates both the state and calculations. Additionally, it requires pool members to take action whenever a commission rate is updated, to prevent any rewards from being lost.

Consider having the commission fixed when a pool is created instead, if it ever needs to be changed. Then, a new pool can be created, and users can switch to the new pool.

Remediation

This issue has been acknowledged by StarkWare Industries. They provided the following response:

> In case of commission change, pool members that respond slowly (in terms of claiming rewards) will lose the saved commission for that response time, and the lost save will remain unclaimed in the pool contract. Fixing this is not trivial, and considering the expected amounts are very small, the fix will not make it but only to the next release.

4.3. Switching delegation to the same pool is allowed

Target	pool.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When a delegator wishes to change their delegation to another pool, they can do so by first signaling their intent to exit the pool for a certain amount. Then they can call `switch_delegation_pool` to immediately change this undelegated amount to a new pool. If they wish to cancel their undelegation, they can call `remove_from_delegation_pool_intent` again with an amount of zero to remove the intent.

The issue is that there is no restriction that the new pool is not the same as the old pool.

Impact

Although there is no immediate impact, switching pools involves keeping track of and syncing up state between various different components. If the delegator wishes to stay with the same pool, they can remove their current delegation intent; there is no need to perform a complex switch to achieve the same result.

Recommendations

Ensure that the new pool is not the same as the old pool.

Remediation

This issue has been acknowledged by StarkWare Industries, and a fix was implemented in commit `c70d5a90` ↗.

4.4. Stale total supply if messages arrive out of order

Target	minting_curve.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When the L1 rewards supplier mints any new coins, it also sends a message to the mining curve on L2 with the updated total supply:

```
// Message updating the total supply, sent by the L1 reward supplier.
#[l1_handler]
fn update_total_supply(ref self: ContractState, from_address: felt252,
    total_supply: Amount) {
    assert_with_err(
        from_address == self.l1_staking_minter_address.read(),
        Error::UNAUTHORIZED_MESSAGE_SENDER
    );
    let old_total_supply = self.total_supply.read();
    self.total_supply.write(total_supply);
    self.emit(Events::TotalSupplyChanged { old_total_supply, new_total_supply:
    total_supply });
}
```

The issue is that if multiple update messages are sent, they may arrive out of order and cause the total supply to be updated with an old value.

Impact

A stale value for the total supply could be used when calculating the yearly mint.

Recommendations

The total supply should only ever increase, so a check could be added to skip any messages that would cause it to be lowered.

Remediation

StarkWare Industries provided the following response:

> In the foreseeable future, `l1_handler` execution out of order is EXTREMELY unlikely and the actual impact on rewards is expected to be minor, even if such a case occurs. We changed the code such that we disregard updates that lower the total supply, to reduce the impact in case this happens.

This issue has been acknowledged by StarkWare Industries, and a fix was implemented in commit `eeba818e` ↗.

4.5. No limit when setting exit window

Target	staking.cairo		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The token-admin role is allowed to change the exit-window period that determines how long a staker must wait between signaling an intent to unstake and actually being able to unstake.

```
fn set_exit_wait_window(ref self: ContractState, exit_wait_window: TimeDelta)
    {
    self.roles.only_token_admin();
    let old_exit_window = self.exit_wait_window.read();
    self.exit_wait_window.write(exit_wait_window);
    self
        .emit(
            ConfigEvents::ExitWaitWindowChanged {
                old_exit_window, new_exit_window: exit_wait_window
            }
        );
}
```

The issue is that there is no limit on what the value of `exit_wait_window` could be, effectively giving the token admin the ability to pause unstaking.

Impact

In practice, the token admin already has a fair amount of control over the protocol, and any calls it makes will need to be performed with extra care. However, having sensible hard limits is a good practice and will help ensure the separation of powers is balanced.

Recommendations

Add a check to ensure that the exit window cannot be made longer than a sensibly chosen hard limit.

Remediation

This issue has been acknowledged by StarkWare Industries, and a fix was implemented in commit `7a3e786d` ↗.

4.6. Rewards supplier can be changed

Target	staking.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The staking contract allows the rewards supply to be set after the contract has been deployed, due to the cyclic dependency between the contracts:

```
fn set_reward_supplier(ref self: ContractState, reward_supplier:
    ContractAddress) {
    self.roles.only_token_admin();
    let old_reward_supplier =
    self.reward_supplier_dispatcher.read().contract_address;
    self
        .reward_supplier_dispatcher
        .write(IRewardSupplierDispatcher { contract_address:
    reward_supplier });
    self
        .emit(
            ConfigEvents::RewardSupplierChanged {
                old_reward_supplier, new_reward_supplier: reward_supplier
            }
        );
}
```

The issue is that there is no check to ensure that the `reward_supplier` has not already been set.

Impact

If the supplier is ever changed, any existing unclaimed rewards could be lost or stuck and would require careful state migration to the new supplier.

Recommendations

Since the intent of the function is to be used as an initializer, a check should be added to ensure that the current `reward_supplier` is not set and the staking contract should no longer take the `reward_supplier` as part of the constructor.

Remediation

This issue has been acknowledged by StarkWare Industries. They provided the following response:

> We consider the risk entailed to be minimal and contained. We will fix in the next release.

4.7. Cannot switch out of finalized pool

Target	staking.cairo		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

When a pool becomes finalized after the staker fully exits, it is no longer possible for delegators to switch from that pool to another. This is because the intent will not be stored on the staking contract when `pool.exit_delegation_pool_intent` is called, and the switch will fail with `MISSING_UNDELEGATE_INTENT`.

Instead, stakers have to fully exit from the pool and redelegate to the new pool instead.

Impact

It is not possible to use `switch_delegation_pool` on a finalized pool. Stakers must instead fully exit and then delegate to the new pool. This behavior could be confusing and will need to be handled specially by any UX designed to interact with the protocol.

Recommendations

If the intent is to not allow switching from a finalized pool, a check to `switch_delegation_pool` should be added along with an appropriate error message.

Remediation

This issue has been acknowledged by StarkWare Industries. They provided the following response:

> This by design. The impact is UX only, as the pool member can exit immediately, and stake on a different pool.

4.8. Periodic mint-cap granularity allows unexpected minting limit

Target	PeriodMintLimit.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `PeriodMintLimit` class sets a cap for the amount of tokens that can be minted within a specific time period, and it applies regardless of anyone's minting limit. It does this by encoding a specific week into a storage slot.

```
uint256 constant MINTING_PERIOD_DURATION = 1 weeks;

function periodAccountingSlot(address token) internal view returns (bytes32) {
    uint256 period_index = block.timestamp / MINTING_PERIOD_DURATION;
    return keccak256(abi.encode(MINTING_PERIOD_DURATION, token,
    period_index));
}
```

Depending on what the cap is intended to limit, it is possible to nearly double the limit in an attack scenario. This is because it does integer division of `block.timestamp`, which will suddenly change the result between some timestamps `T` and `T+1`.

Impact

At the boundary of a new week, it is possible to mint the remainder of `PERIOD_MINT_CAP` at timestamp `T` and then another full `PERIOD_MINT_CAP` in the next block, `T+1`. This means that it is technically possible to mint `2 * PERIOD_MINT_CAP` within seconds and not a full week, as possibly intended. There is not necessarily any security impact in the current implementation, and this finding is just informational in nature.

Recommendations

From the code's documentation and comments, it is not clear what this minting cap is supposed to limit or how the current value for `PERIODIC_MINT_CAP` was derived. If the intention is to limit the gain from malicious interactions with the contract, until the system can be paused, a system that looks at the past week as a sliding window might be more beneficial to implement. However, it would also be significantly more complex.

Remediation

This issue has been acknowledged by StarkWare Industries. They provided the following response:

> We are aware of the design and its limitation, and don't consider it an issue. The granularity of minting and quota were reviewed with taking into account the overall context of deployment and usage, and with consideration of time and effort, and the design we ended up with was considered reasonable given all these considerations.

5. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

5.1. Circular dependencies of the constructors

The constructor of the Staking contract requires the address of the Reward Supplier contract. However, the constructor of the Reward Supplier contract also requires the address of the Staking contract. In addition, the constructor of the Minting Curve contract requires the address of the Staking contract, and the Reward Supplier contract — whose address is required by the Staking contract — requires the address of the Minting Curve contract.

The deployer would have to precompute the addresses of these contracts in order to successfully deploy them.

5.2. Trusted sequencer assumption

The block timestamp (seconds since Unix epoch) fetched by the system call `get_block_timestamp()` is provided by the sequencer of the block.

The documentation from Cairo ↗ states the following:

> Presently, the result of `get_block_timestamp()` is not enforced by the StarkNet OS or Core contract (i.e., the sequencer may choose an arbitrary timestamp). In the future, some restrictions on the new timestamp will be added. Also note that the block timestamp is the time at the beginning of the block creation, which can differ significantly from the time the block is accepted on L1.

In the future, when stakers are running their own sequencers, this could become a problem. Stakers could maliciously modify the timestamp in order to inflate their rewards or to skip the waiting time for unstaking.

StarkWare Industries states that

> You are correct this design assumes a trusted sequencer. It's ok since in the decentralized starknet the rewards distribution model will be completely different, the rewards will be distributed according to actions you do, not [how much] time passed.

There are additional points of concern in the presence of malicious sequencers.

It is also possible for the sequencer to drop arbitrary L1 → L2 messages, which could affect token transfers via StarkGate. Any dropped messages will cause the reward supplies `l1_pending_requested_amount` to be incorrect and to be waiting for tokens that are never coming. This could be manually fixed by sending the missing tokens again from a different source, but there is no way to recover the locked tokens on the L1.

In a similar vein, the TimeDelta implementation in time.cairo is backed by an unsigned integer, which can overflow or underflow, producing a crash that reverts the transaction. If the transaction was a result of, for example, token transfers, a malicious sequencer could make a valid L1 transfer fail on L2, potentially locking funds.

6. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

> This threat model was created during the first primary audit period, and due to major refactoring, not all of the comments still apply. Additionally, extensive testing has been added both between the audit periods and after the second audit period.

6.1. Module: minting_curve.cairo

Function: `update_total_supply(ref self: ContractState, from_address: felt252, total_supply: felt252)`

This sets the new total supply from L1.

Inputs

- `from_address`
  - **Validation**: Automatically set. Must be equal to the minter address.
  - **Impact**: The L1 address that initiated the call.
- `total_supply`
  - **Validation**: Must fit inside a `u128`.
  - **Impact**: The new total supply to store.

Branches and code coverage (including function calls)

Intended branches

- Update supply successfully.
  - [ ] Test coverage

Negative behavior

- The caller is not the staking minter.
  - [ ] Negative test
- The value is larger than 2^128.
  - [ ] Negative test

6.2. Module: operator.cairo

Function: `change_operational_address(operational_address)`

This changes the operational address for a given staker.

The function can only be called when the global state is unpaused. If the index has not been updated for `MIN_DAYS_BETWEEN_INDEX_UPDATES` days, it will be updated before the rest of the function runs.

Because of the proxied nature of this call, the target contract cannot fetch the caller address directly to ascertain which user is trying to stake. Instead, this is done by calling `get_tx_info().account_contract_address`. As mentioned in Finding 3.2. ↗, this is dangerous.

Inputs

- `operational_address`
  - **Validation**: None. Can even overwrite existing operational addresses.
  - **Impact**: The new operational address the staker should be associated with.

Branches and code coverage (including function calls)

Intended branches

- Change operational address successfully.
  - ☑ Test coverage

Negative behavior

- Staker does not exist.
  - ☑ Negative test
- Called during pause.
  - ☐ Negative test

Function: `change_reward_address(reward_address)`

This changes the reward address for the calling staker.

If global whitelisting is enabled, the caller must be in the whitelist. The function proxies into the staking contract, calling the function with the same name and parameters. This is done because the staking contract only permits the operator contract to call into it. For this reason, the proxied function will be described here instead.

The function can only be called when the global state is unpaused. If the index has not been updated for `MIN_DAYS_BETWEEN_INDEX_UPDATES` days, it will be updated before the rest of the function runs.

Because of the proxied nature of this call, the target contract cannot fetch the caller address directly to ascertain which user is trying to stake. Instead, this is done by calling `get_tx_info().account_contract_address`. As mentioned in Finding 3.2. ↗, this is dangerous.

Inputs

- `reward_address`
    - **Validation**: None.
    - **Impact**: The new reward address for the calling staker.

Branches and code coverage (including function calls)

Intended branches

- Successfully change the reward address.
    - ☑ Test coverage

Negative behavior

- Staker does not exist.
    - ☑ Negative test
- Global pause is active.
    - ☐ Negative test

Function: `claim_rewards(staker_address)`

This claims staker rewards. Any rewards (minus commission) will be sent to the reward address. The original caller must either be the `staker_address` itself or the `staker_info.reward_address`.

If global whitelisting is enabled, the caller must be in the whitelist. The function proxies into the staking contract, calling the function with the same name and parameters. This is done because the staking contract only permits the operator contract to call into it. For this reason, the proxied function will be described here instead.

The function can only be called when the global state is unpaused. If the index has not been updated for `MIN_DAYS_BETWEEN_INDEX_UPDATES` days, it will be updated before the rest of the function runs.

Because of the proxied nature of this call, the target contract cannot fetch the caller address directly to ascertain which user is trying to stake. Instead, this is done by calling `get_tx_info().account_contract_address`. As mentioned in Finding 3.2. ↗, this is dangerous.

Inputs

- `staker_address`
    - **Validation**: Has to be equal to the original caller, unless the caller is equal to the reward address fetched from the information associated with the staker.
    - **Impact**: The address of the staker to claim rewards for.

Branches and code coverage (including function calls)

Intended branches

- Claim rewards as the staker address.
  - ☑ Test coverage
- Claim rewards as the reward address.
  - ☐ Test coverage
- Claim zero rewards.
  - ☐ Test coverage

Negative behavior

- Claim rewards as neither the staker or reward address.
  - ☑ Negative test
- Claim rewards as a staker that does not exist.
  - ☑ Negative test

Function call analysis

- `claim_rewards -> calculate_rewards(staker_info)`
  - **External/Internal?** Internal.
  - **Argument control?** `staker_info` is based on the staking info of `staker_address`. Not directly controllable.
  - **Impact** The stored information about the staker to calculate the reward for.

Function: `increase_stake(staker_address, amount)`

This increases the stake for a single staker. It cannot be called if the staker has initiated an unstake. The original caller must either be the `staker_address` itself or the `staker_info.reward_address`.

If global whitelisting is enabled, the caller must be in the whitelist. The function proxies into the staking contract, calling the function with the same name and parameters. This is done because the staking contract only permits the operator contract to call into it. For this reason, the proxied function will be described here instead.

The function can only be called when the global state is unpaused. If the index has not been updated for `MIN_DAYS_BETWEEN_INDEX_UPDATES` days, it will be updated before the rest of the function runs.

Because of the proxied nature of this call, the target contract cannot fetch the caller address directly to ascertain which user is trying to stake. Instead, this is done by calling `get_tx_info().account_contract_address`. As mentioned in Finding 3.2. ↗, this is dangerous.

Inputs

- `staker_address`

  - **Validation**: Has to be equal to the original caller, unless the caller is equal to the reward address fetched from the information associated with the staker.
  - **Impact**: The address of the staker to increase the stake for.
- `amount`
  - **Validation**: None.
  - **Impact**: The amount to increase the stake with.

Branches and code coverage (including function calls)

Intended branches

- Incease stake from the staker address.
  - ☑ Test coverage
- Increase stake from the reward address.
  - ☑ Test coverage
- Stake zero amount.
  - ☑ Test coverage

Negative behavior

- Staker address does not exist.
  - ☑ Negative test
- Unstaking is in progress.
  - ☑ Negative test
- Caller is not staker nor reward address.
  - ☑ Negative test
- Global pause is active.
  - ☐ Negative test
- Whitelisting is active and user is not whitelisted.
  - ☐ Negative test

Function call analysis

- `increase_stake -> calculate_rewards(staker_info)`
  - **External/Internal?** Internal.
  - **Argument control?** `staker_info` is based on the staking information of `staker_address`. Not directly controllable.
  - **Impact**: The stored information about the staker to calculate the reward for.
- `increase_stake -> add_to_total_stake(amount)`
  - **External/Internal?** Internal.
  - **Argument control?** `amount` is fully controllable, given the caller has the balance.
  - **Impact**: The amount to add to the total stake.

Function: `set_open_for_delegation(commission)`

If the initial caller (staker) does not already have a delegation pool, this function deploys a delegation pool contract on behalf of the staker. The pool contract's `staking_contract` field is set to the staking contract address, and its `token_address` is copied from `erc20_dispatcher.contract_address`, while `commission` is provided directly. The commission cannot currently be updated once set.

If global whitelisting is enabled, the caller must be in the whitelist. The function proxies into the staking contract, calling the function with the same name and parameters. This is done because the staking contract only permits the operator contract to call into it. For this reason, the proxied function will be described here instead.

The function can only be called when the global state is unpaused. If the index has not been updated for `MIN_DAYS_BETWEEN_INDEX_UPDATES` days, it will be updated before the rest of the function runs.

Because of the proxied nature of this call, the target contract cannot fetch the caller address directly to ascertain which user is trying to stake. Instead, this is done by calling `get_tx_info().account_contract_address`. As mentioned in Finding 3.2. ↗, this is dangerous.

Inputs

- `commission`
  - **Validation**: Must be less than `COMMISSION_DENOMINATOR`.
  - **Impact**: The commission to provide to the pool, as a fraction denominated by `COMMISSION_DENOMINATOR`.

Branches and code coverage (including function calls)

Intended branches

- Deploy a delegation pool.
  - ☑ Test coverage

Negative behavior

- Commission is too high.
  - ☑ Negative test
- Staker does not exist.
  - ☑ Negative test
- Staker already has a pool.
  - ☑ Negative test

Function call analysis

- `set_open_for_delegation -> deploy_delegation_pool_contract(staker_address, staking_contract, token_address, commission)`

- **External/Internal?** Internal.
- **Argument control?** Only `commission` is controllable.
- **Impact**: Deploys the delegation pool and returns its contract address. The staking contract will be the admin of the new pool.

Function: `stake(reward_address, operational_address, amount, pooling_enabled, commission)`

This adds a new staker to the stake. If global whitelisting is enabled, the caller must be in the whitelist. The function proxies into the staking contract, calling its `stake(...)` function with the same parameters, since the staking contract only permits the operator contract to call into it. For this reason, the proxied function will be described here instead.

The function can only be called when the global state is unpaused. If the index has not been updated for `MIN_DAYS_BETWEEN_INDEX_UPDATES` days, it will be updated before the rest of the function runs.

Because of the proxied nature of this call, the target contract cannot fetch the caller address directly to ascertain which user is trying to stake. Instead, this is done by calling `get_tx_info().account_contract_address`. As mentioned in Finding 3.2. ↗, this is dangerous.

Inputs

- `reward_address`
    - **Validation**: None.
    - **Impact**: The address that will receive the rewards for the stake.
- `operational_address`
    - **Validation**: Cannot already exist in the `operational_address_to_staker_address` mapping.
    - **Impact**: The operational address associated with the stake.
- `amount`
    - **Validation**: Must be larger than the minimum stake amount.
    - **Impact**: The amount to stake. This is transferred from the original caller address to the staking contract.
- `pooling_enabled`
    - **Validation**: None.
    - **Impact**: Decides if pooling should be enabled or not. Generates a pooling contract instance if enabled.
- `commission`
    - **Validation**: Cannot be larger than 10,000, which is 100%.
    - **Impact**: The commission to give to the pool.

Branches and code coverage (including function calls)

Intended branches

- Stake without a pool.
  - ☑ Test coverage
- Stake with a pool.
  - ☑ Test coverage

Negative behavior

- Staker already exists.
  - ☑ Negative test
- Operational address already exists.
  - ☑ Negative test
- Commission is out of range.
  - ☑ Negative test
- Stake is too low.
  - ☑ Negative test
- Stake during global pause.
  - ☑ Negative test

Function call analysis

- `stake -> deploy_delegation_pool_contract(staker_address, staking_contract, token_address, commission)`
  - **External/Internal?** Internal. Only called if `pooling_enabled` is true.
  - **Argument control?** Only commission is controllable.
  - **Impact**: Deploys a pool with the given commision. It can be updated later.
- `stake -> add_to_total_stake(amount)`
  - **External/Internal?** Internal.
  - **Argument control?** Amount is controllable.
  - **Impact**: The amount staked, which is subsequently added to the total stake count.

Function: `unstake_action(staker_address)`

This finalizes an ongoing unstake intent. Can be called by any user — but only if the waiting period has passed (i.e., `get_block_timestamp() >= unstake_time`).

If global whitelisting is enabled, the caller must be in the whitelist. The function proxies into the staking contract, calling the function with the same name and parameters. This is done because the staking contract only permits the operator contract to call into it. For this reason, the proxied function will be described here instead.

The function can only be called when the global state is unpaused. If the index has not been updated for `MIN_DAYS_BETWEEN_INDEX_UPDATES` days, it will be updated before the rest of the function runs.

Inputs

- `staker_address`
    - **Validation**: Must have a valid unstake intent associated with it.
    - **Impact**: The address of the staker that will finalize its unstaking intent.

Branches and code coverage (including function calls)

Intended branches

- Unstake successfully after the waiting period.
    - [x] Test coverage

Negative behavior

- Unstake an invalid address (missing intent).
    - [ ] Negative test
- Unstake before the waiting period has passed.
    - [ ] Negative test

Function call analysis

- `unstake_action -> transfer_to_pool_when_unstake(staker_address, staker_info)`
    - **External/Internal?** Internal.
    - **Argument control?** `staker_address` is fully controlled. `staker_info` is fetched based on the `staker_address`.
    - **Impact**: Transfers unclaimed rewards to the pool.
- `unstake_action -> remove_staker(staker_address, staker_info)`
    - **External/Internal?** Internal.
    - **Argument control?** `staker_address` is fully controlled. `staker_info` is fetched based on the `staker_address`.
    - **Impact**: Removes the staker by deleting its associated `staker_info` struct and its operational address from the global mapping.

Function: `unstake_intent()`

This signals that the caller wants to unstake in the future. It reads the value of `exit_wait_window` and adds this to the current block timestamp. The sum becomes the `unstake_time`, which is the timestamp where someone can call `unstake_action()` to finalize the unstaking. It is not allowed to signalize an unstake intent while one is in progress.

If global whitelisting is enabled, the caller must be in the whitelist. The function proxies into the staking contract, calling the function with the same name and parameters. This is done because the staking contract only permits the operator contract to call into it. For this reason, the proxied function will be described here instead.

The function can only be called when the global state is unpaused. If the index has not been updated for `MIN_DAYS_BETWEEN_INDEX_UPDATES` days, it will be updated before the rest of the function runs.

Because of the proxied nature of this call, the target contract cannot fetch the caller address directly to ascertain which user is trying to stake. Instead, this is done by calling `get_tx_info().account_contract_address`. As mentioned in Finding 3.2. ↗, this is dangerous.

Branches and code coverage (including function calls)

Intended branches

- Unstake intent is successful.
  - ☑ Test coverage

Negative behavior

- Staker does not exist.
  - ☑ Negative test
- Staker is already unstaking.
  - ☑ Negative test

Function call analysis

- `unstake_intent -> remove_from_total_stake(amount)`
  - **External/Internal?** Internal.
  - **Argument control?** `amount` has no direct control. It is based on the amount the staker owns and has in its pool.
  - **Impact**: The amount of tokens to remove from the total stake.

6.3. Module: reward_supplier.cairo

Function: `on_receive(l2_token, amount, depositor, message)`

This processes the assets received through the StarkGate protocol. Should only be callable by StarkNet but is missing this check, as described in Finding 3.3. ↗.

Inputs

- `l2_token`
  - **Validation**: None.

    - **Impact**: Unused.
- `amount`
    - **Validation**: Has to fit in a `u128` type.
    - **Impact**: The amount that was received.
- `depositor`
    - **Validation**: None.
    - **Impact**: Unused.
- `message`
    - **Validation**: None.
    - **Impact**: Unused.

Branches and code coverage (including function calls)

Intended branches

- Successfully receive multiple messages.
    - [x] Test coverage

Negative behavior

- Caller is not Starknet.
    - [ ] Negative test

7. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Starknet mainnet.

During our assessment on the scoped Starknet Staking contracts, we discovered 15 findings. One critical issue was found. Three were of high impact, four were of medium impact, four were of low impact, and the remaining findings were informational in nature.

7.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.