Security Review Report
NM-0337 Starkstake_

NETHERMIND
SECURITY

(Jan 20, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for Starkstake_ smart contracts. *Starkstake_* allows you to stake your Starknet tokens, earn rewards, and receive *stSTRK* tokens. *stSTRK* is a liquid staking derivative that represents your staked assets and allows you to remain flexible, trade, or use them in DeFi while still earning staking rewards.

The audited code comprises 1429 lines of code written in the Cairo language. The audit focuses on the implementation of *stStark* token, the main contract for managing interactions, and the contracts representing delegators.

The audit was performed using (a) manual analysis of the codebase and (b) creation of test cases. **Along this document, we report** fourteen points of attention, where one is classified as `Critical`, one is classified as `High`, one is classified as `Medium`, two are classified as `Low`, and nine are as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

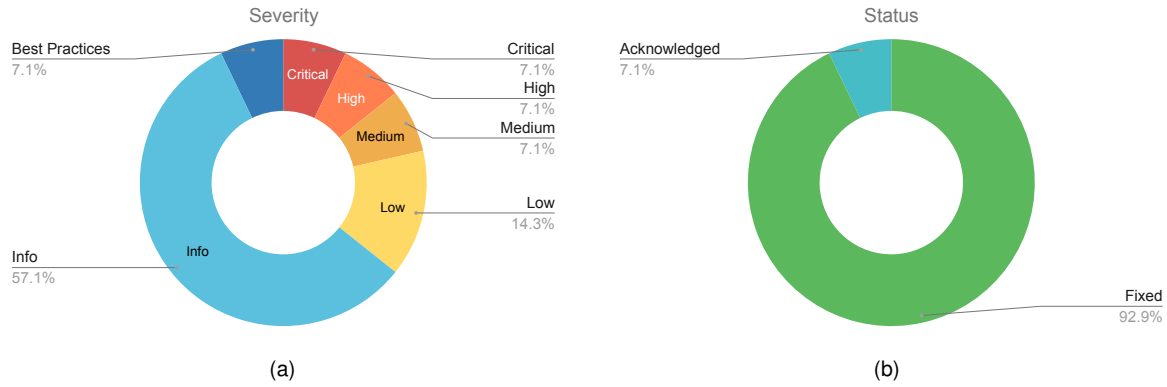


Fig. 1: Distribution of issues: Critical (1), **High** (1), **Medium** (1), **Low** (2), **Undetermined** (0), **Informational** (8), **Best Practices** (1). **Distribution of status: Fixed** (13), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	October 31, 2024
Final Report	January 20, 2025
Repository	starkstake_
Commit	f9bf6c0db652cc561a1d7dac491f5857fe051fc8
Final Commit	3e42e236a24be476e9b40610f74a7986379d19bd
Documentation	Not public
Documentation Assessment	High
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/lib.cairo	26	2	7.7%	1	29
2	src/contracts/delegator.cairo	146	65	44.5%	38	249
3	src/contracts/stake_stark.cairo	575	273	47.5%	128	976
4	src/contracts/stSTRK.cairo	267	200	74.9%	63	530
5	src/utils/constants.cairo	11	10	90.9%	3	24
6	src/components/access_control.cairo	107	5	4.7%	21	133
7	src/interfaces/i_stake_stark.cairo	92	19	20.7%	20	131
8	src/interfaces/i_delegator.cairo	24	5	20.8%	5	34
9	src/interfaces/i_starknet_staking.cairo	129	18	14.0%	14	161
10	src/interfaces/i_stSTRK.cairo	52	18	34.6%	8	78
	Total	**1429**	**615**	**43.0%**	**301**	**2345**

3 Summary of Issues

	Finding	Severity	Update
1	The `is_in_pool` variable should be set to false if all the funds are withdrawn by the delegator	Critical	Fixed
2	Deposit and Withdrawal actions revert permanently if all delegators are used	High	Fixed
3	The `get_tx_info().account_contract_address` can be exploited if user interacts with malicious contract	Medium	Fixed
4	Incorrect best fit algorithm for selecting delegator for withdrawal	Low	Fixed
5	The `STRK` token deposits via the `stSTRK` contract will revert in specific scenarios	Low	Fixed
6	Event emitted with incorrect parameter	Info	Fixed
7	Incorrect value assigned to `BURNER_ROLE` and `OPERATOR_ROLE` constant	Info	Fixed
8	Missing `pause` and `unpause` functions	Info	Fixed
9	Rewards distribution is not fully fair	Info	Acknowledged
10	The `PAUSER_ROLE` isn't granted to any address	Info	Fixed
11	The `burn(...)` function doesn't assert with `BURNER_ROLE`	Info	Fixed
12	The `delegate(...)` function uses an outdated delegation pool interface	Info	Fixed
13	The `get_withdrawable_amount(...)` and `get_all_withdrawal_requests(...)` functions have incorrect implementations	Info	Fixed
14	The `last_processing_time` and `last_reward_claim_time` state variables could be emitted as an event	Best Practices	Fixed

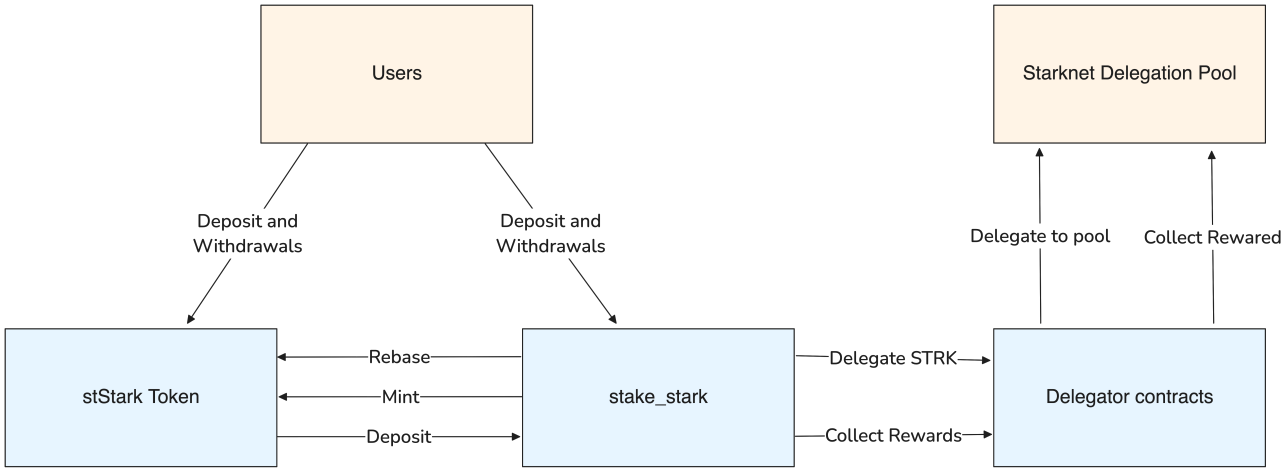
4 System Overview

*starkstake_* is a liquid staking protocol built on Starknet, allowing users to stake their *STRK* assets while maintaining liquidity. In traditional staking models, staked assets are locked, limiting liquidity. However, *starkstake_* issues a liquid staking token called *stSTRK* when users stake their *STRK*, enabling them to use these tokens across DeFi protocols.

By leveraging *starkstake_*, users can benefit from staking rewards without losing the ability to actively manage and use their assets. The protocol is implemented using smart contracts on Starknet, ensuring both high security and transparency.

Users interact directly with the *stSTRK* token or the *stake_stark* contracts. In both cases, users can execute deposit and withdrawal operations, giving two different entry points to the protocol. Users deposit *STRK* tokens and receive *stSTRK* tokens that increase in value over time. The value of the *stSTRK* token is defined by the amount of *STRK* held by the protocol and the amount of *stSTRK* total supply. The protocol behaves similarly to an ERC-4626 vault where the underlying token is *STRK*.

The received *STRK* tokens are staked in Starknet in order to generate rewards. The staking occurs through multiple delegator contracts, deployed by the *Starkstake_* protocol, that join a staking pool also managed by the protocol itself.



(c)

4.1 Deposits

To execute a deposit, users call the *deposit(...)* or *mint(...)* functions. With this action, users will send *STRK* tokens to the protocol and receive *stSTRK* tokens in exchange. The deposited funds will stay in the *stake_stark* contract until the deposits are processed. The amount of *stSTRK* minted depends on the current ratio of *STRK* and *stSTRK* tokens.

4.2 Withdrawals

The withdrawal of `stSTRK` tokens is a two-step process. To receive the `STRK` tokens, the users must first request the withdrawal, wait for the withdrawal window period to pass, and finally withdraw the `STRK` tokens. Users can submit the requests by calling the `request_withdrawal(...)` function from the `stake_stark` contract or the `withdraw(...)` and `redeem(...)` functions from the `stSTRK` contract. By requesting the withdrawal, users burn their `stSTRK` tokens and start the withdrawal window period of at least 21 days. The request is assigned an ID. The amount of assets to be withdrawn is tied to the user's address and ID and stored in a mapping, from where it will be retrieved once the withdrawal window period ends. Once the request is ready to be withdrawn, the user can call the `withdraw(...)` function on the `stake_stark` contract to receive the `STRK` tokens.

Users' withdrawal requests and deposit operations are batched for efficiency reasons. The protocol controlled *Operator* will simultaneously process the withdrawal requests and deposits. The following section explains the batch processing in greater detail.

4.3 Batch Processing

Delegating, undelegating, and staking reward collection must be performed regularly in the *Starkstake_* protocol. At regular intervals (at least once a day), the account with the *Operator* role controlled by the *Starkstake_* protocol will call the `process_batch(...)` function to fulfill these duties. Given the fact that some users want to deposit `STRK` tokens, while others might want to withdraw their tokens, the `process_batch(...)` function optimizes the process to avoid unnecessary delegation and undelegation operations. If the deposits are higher than the withdrawals, only the difference between the two will be delegated, while the rest will await withdrawal by the user. In the opposite case, only the difference will be undelegated from the Delegation Pool.

The `process_batch(...)` function also collects the `STRK` token rewards from the delegators and distributes them in the form of a `stSTRK` rebase.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] The `is_in_pool` variable should be set to false if all the funds are withdrawn by the delegator

File(s): src/contracts/delegator.cairo

Description: The `process_withdrawal(...)` function is called to process the withdrawal request after the `withdrawal_window_period` has passed for a particular delegator. It further calls `pool.exit_delegation_pool_action(...)` to withdraw the funds from the pool.

```
fn process_withdrawal(ref self: ContractState) -> u256 {
    // ...
    let pool = IPoolDispatcher {contract_address: self.pool_contract.read()};

    // @audit Withdraw funds from the pool.
    let withdrawn_amount: u256 = pool
        .exit_delegation_pool_action(get_contract_address())
        .into();

    // ...
    // @audit Send the withdrawn tokens to the user.
    strk_token.transfer(self.stake_stark.read(), withdrawn_amount);

    // ...
}
```

The `pool.exit_delegation_pool_action(...)` function will remove the delegator from the pool if the delegator has withdrawn all the delegated amounts.

```
fn exit_delegation_pool_action(
    ref self: ContractState, pool_member: ContractAddress
) -> Amount {
    let mut pool_member_info = self.get_pool_member_info(:pool_member);
    // ...

    if pool_member_info.amount.is_zero() {
        // @audit Delegator will be entirely removed if it no longer
        // has any delegated funds.
        self.remove_pool_member(:pool_member);
    } else {
        // @audit Otherwise, update delegator info. The delegator stays
        // in the pool. Only this case is considered by process_withdrawal(...).
        pool_member_info.unpool_time = Option::None;
        self.pool_member_info.write(pool_member, Option::Some(pool_member_info));
    }
    // ...
}
```

The problem present in the `process_withdrawal(...)` function is that it does not consider the case where the delegator is removed entirely from the pool. Because of that, the `is_in_pool` variable will still be set to `true` even though the delegator is no longer in the pool.

Later, when the `delegate(...)` function is called by the `StakeStark` contract to deposit the funds in the pool, since the delegator is still considered to be in the pool, the `add_to_delegation_pool(...)` function will be called instead of the `enter_delegation_pool(...)`.

```
fn delegate(ref self: ContractState, amount: u256) {
    // ...
    if !self.is_in_pool.read() {
        // @audit If the delegator is not in the pool, it must first be registered as a pool member.
        let success = pool
            .enter_delegation_pool(self.stake_stark.read(), amount_u128);
        assert(success, 'Failed to enter delegation pool');
        self.is_in_pool.write(true);
    } else {
        // @audit This function will be called instead since `is_in_pool` is set to true.
        // The delegation will fail since the delegator is not a pool member.
        let new_total = pool.add_to_delegation_pool(get_contract_address(), amount_u128);
        // ...
    }
    // ...
}
```

As a result, the `add_to_delegation_pool(...)` function will revert since the delegator is no longer a pool member, and the delegation process will fail.

During the deposit process, the Stake Stark protocol balances the delegations across all the available delegators. Whenever the batch of deposits is processed, a delegator with the least stake will be chosen to fulfill the delegation request. Since the delegator mentioned in the previous paragraph is still considered available and has a total stake of `0`, it will be chosen every time as the best fit to fulfill the delegation.

Since its `delegate(...)` function will revert due to the abovementioned issue, no funds will be delegated. Thus, even if only one delegator is affected by this problem, no subsequent deposit will be processed and delegated to the delegation pool.

Recommendation(s): Consider changing the logic to set the `is_in_pool` variable to `false` whenever the delegator's `total_stake` reaches `0` and it is no longer a pool member.

Status: Fixed.

Update from the client: fixed. commit hash: c1b14ab.

6.2 [High] Deposit and Withdrawal actions revert permanently if all delegators are used

File(s): src/contracts/stake_stark

Description: The `process_batch(...)` function executes multiple critical operations for the protocol. The function settles the pending `withdrawal` and `deposit` operations, finishes the current un-delegation actions, and accrues rewards.

```
fn process_batch(ref self: ContractState) {
    self.pausable.assert_not_paused();
    self.access_control.assert_only_role(OPERATOR_ROLE);
    self.reentrancy_guard.start();

    self._process_batch();
    self._delegator_withdraw();

    let total_rewards = self._collect_rewards_from_delegators();

    if total_rewards > 0 {
        self._distribute_rewards(total_rewards);
    }

    self.reentrancy_guard.end();
}
```

The `_process_batch(...)` function calls `_delegate_to_available_delegator(...)` or `_request_withdrawal_from_available_delegator(...)` depending on the pending amount of funds to be withdrawn or deposited. Both of these functions require that at least one delegator is available.

A problem arises because the `_delegator_withdraw(...)` must be executed in order to make delegators available again. If all the delegators are unavailable at some point, the `_delegator_withdraw(...)` function must be executed to make delegators available again. However, because there are no available delegators to execute the `_delegate_to_available_delegator(...)` or `_request_withdrawal_from_available_delegator(...)`, the call to the `process_batch(...)` function will fail, leaving the protocol in a state where `Withdrawals` and `Deposits` are not available.

The protocol can only leave this state if the pending amounts to deposit and withdraw are even.

Recommendation(s): Consider changing the order of the operations `_process_batch(...)` and `_delegator_withdraw()` in the `process_batch(...)` function. Alternatively, add a function to execute `_delegator_withdraw(...)` independently.

Status: Fixed.

Update from the client: fixed. hash: 1e9cf7e.

6.3 [Medium] The `get_tx_info().account_contract_address` can be exploited if user interacts with malicious contract

File(s): src/contracts/stake_stark.cairo

Description: The `deposit(...)`, `_process_withdrawal_request(...)` and `_process_withdrawals(...)` functions use `get_tx_info().account_contract_address` to get the `caller` address, which returns the address from which the transaction originated. The `deposit(...)` function is shown below:

```
fn deposit(ref self: ContractState, amount: u256, receiver: ContractAddress) -> u256 {
    // ...
    let caller = get_tx_info().account_contract_address;
    // ...
    // @audit Transfer the tokens from the account that originated the transaction.
    assert(
        strk_dispatcher.transfer_from(caller, get_contract_address(), amount),
        'Transfer failed'
    );

    let shares = if receiver.is_zero() {
        stSTRK.mint(amount, caller)
    } else {
        // @audit Mint the shares to the receiver passed in as argument to this function.
        stSTRK.mint(amount, receiver)
    };
    // ...
}
```

Consider a case where a user has infinite `STRK` token approval to the `stake_stark` contract and interacts with any malicious contract. The malicious contract can call the `deposit(...)` function with `receiver` as itself. Now, as `get_tx_info().account_contract_address` will return the user's address, the malicious contract will receive the shares, which will be minted using the user's funds.

The `request_withdrawal(...)` and `withdraw(...)` functions have similar issues, but for these functions, there isn't any `receiver` parameter. Thus, in this case, a malicious contract can withdraw the user's funds without the user's consent, and the user will stop getting the rewards. However, funds received from the withdrawal will be sent to the user only, so there will be no loss of funds by withdrawal, unlike deposits.

Recommendation(s): Consider replacing the instances of `get_tx_info().account_contract_address` with `get_caller_address()` to access the user's address directly. Since the `stSTRK` contract also calls `deposit(...)`, `request_withdrawal(...)`, and `withdraw(...)` functions, one of the possible solutions is to add a user's address as a parameter to already existing functions and only allow for it to be different than the address returned by `get_caller_address()` if the caller is `stSTRK.cairo`.

Status: Fixed.

Update from the client: fixed. commit hash: 02e022f.

Update from Nethermind: The fix for `request_withdrawal(...)` would break the `redeem(...)` and `withdraw(...)` functions from `stSTRK`. Also, `execute_withdrawals` still uses `tx.origin`

Update from the client: fixed. commit hash: 5f6ac8e, 3274ccf.

Update from Nethermind: This was fixed by a redesign of the architeture that removes the deposit/withdraw functionalities from the `stSTRK` token.

6.4 [Low] Incorrect best fit algorithm for selecting delegator for withdrawal

File(s): src/contracts/stake_stark.cairo

Description: The `_request_withdrawal_from_available_delegator(...)` function is called to request withdrawals from available delegators to fulfill the given amount. It has two loops which go through each of the available delegators and the first one try to select a delegator which is a best fit i.e. a delegator with least delegated stake which can fullfil the withdrawal request. If there isn't any delegator which can fullfil the given amount then in the second loop, multiple delegators are selected to fullfil the request. But there's a problem in the implementation of best fit algorithm where if first delegator considered in the loop doesn't have sufficient delegated stake then it will be considered as best fit and no other delegator will be taken into consideration. Thus, even if a best fit exists, it won't be selected. The impact of this issue is that multiple delegators would become unavailable which might distrupt withdrawal process i.e. more delay might occur for withdrawal.

```
fn _request_withdrawal_from_available_delegator(ref self: ContractState, amount: u256) {
    let mut remaining_amount = amount;

    // First pass: find the best fit delegator
    let mut i: u8 = 0;
    let mut best_fit_index: u8 = 0;
    let mut best_fit_amount: u256 = 0;

    while i < self.num_delegators.read() {
        if self._is_delegator_available(i) {
            let delegator = IDelegatorDispatcher { contract_address: self.delegators.read(i) };

            let delegator_stake = delegator.get_total_stake();

            // @audit Consider delegators with following stakes: [1,20,40]
            // and amount requested for withdrawal be 30
            // The best fit in this case would be the first element
            // whereas it should be third element.
            if delegator_stake > 0 {
                if delegator_stake >= remaining_amount
                    && (best_fit_amount == 0 || delegator_stake < best_fit_amount) {
                    best_fit_index = i;
                    best_fit_amount = delegator_stake;
                } else if best_fit_amount == 0 {
                    best_fit_index = i;
                    best_fit_amount = delegator_stake;
                }
            }
        }
        i += 1;
    };

    // Second pass: process withdrawal requests
    i = 0;
    while remaining_amount > 0 && i < self.num_delegators.read() {
        // ...
    };

    assert(remaining_amount == 0, 'Insufficient available funds');
}
```

Recommendation(s): Consider revisiting the best fit algorithm to be in accordance with the expected strategy.

Status: Fixed.

Update from the client: fixed. hash: b1f15da.

6.5 [Low] The STRK token deposits via the stSTRK contract will revert in specific scenarios

File(s): src/contracts/stSTRK.cairo

Description: Users have two entry points to start participating in the liquid staking process in the Stake Stark protocol. To deposit the STRK tokens, users can call the deposit(...) function directly on either the main StakeStark contract or the stSTRK token contract. Under the hood, the deposit(...) function defined in the liquid staking token contract calls the deposit(...) from the StakeStark contract. This means that it shouldn't matter which function the user chooses since, eventually, they will follow the same execution path. This, however, is not the case since both functions compute the expected stSTRK token shares differently. The deposit(...) function from the stSTRK contract uses the preview_deposit(...) function, while the deposit(...) from the StakeStark contract uses the preview_mint(...). Both preview functions take the amount of assets as an argument and return the computed shares amount, but they differ in the rounding direction used in the computation process.

The preview_deposit(...) function conforms with the ERC4626 standard and correctly rounds down.

```
// @audit This function is used in the StakeStark deposit(...)
fn preview_deposit(self: @ContractState, assets: u256) -> u256 {
    // @audit This rounds down.
    self.convert_to_shares(assets)
}

fn convert_to_shares(self: @ContractState, assets: u256) -> u256 {
    if self.total_assets() == 0 {
        assets * INITIAL_SHARES_PER_ASSET
    } else {
        // @audit Rounding down in the preview_deposit(...) happens here.
        (assets * self.erc20.total_supply()) / self.total_assets()
    }
}
```

However, the preview_mint(...) function deviates from the ERC4626 standard since, instead of shares, it accepts assets as an input parameter. The function's logic follows the standard and rounds up, but since the computation is based on assets instead of shares, this leads to an incorrect behavior. In situations when division results in rounding, the preview_mint(...) function will return a higher shares amount than the preview_deposit(...) function for the same amount of assets.

```
// @audit-issue The preview_mint function in ERC4626 takes
// shares instead of assets. Because of that, the rounding direction
// in the computation below is incorrect. It rounds up instead of down.
fn preview_mint(self: @ContractState, assets: u256) -> u256 {
    if self.erc20.total_supply() == 0 {
        assets * INITIAL_SHARES_PER_ASSET
    } else {
        // @audit This rounds up.
        (assets * self.erc20.total_supply() + self.total_assets() - 1)
            / self.total_assets()
    }
}
```

Whenever this discrepancy happens, the deposit(...) function from the stSTRK contract will revert.

```
fn deposit(
    ref self: ContractState, assets: u256, receiver: ContractAddress
) -> u256 {
    // ...
    // @audit shares - rounded down
    let shares = self.preview_deposit(assets);
    // ...
    // @audit shares rounded up
    let minted_shares = stake_stark.deposit(assets, receiver);

    // @audit-issue Whenever there is a division with a remainder
    // and rounding is triggered, this will revert.
    assert(minted_shares == shares, 'Shares mismatch');
    // ...
}
```

The impact of this issue is limited to the inconvenience that the users might face in case of unexpected reverts.

Recommendation(s): Consider revisiting the logic in the preview_mint(...) function so that it follows the ERC4626 standard.

Status: Fixed.

Update from the client: fixed. commit hash: df7f3ff

Update from Nethermind: The `preview_mint(...)` function should round-up

Update from the client: fixed. commit hash: ab96ac4

Update from Nethermind: This was fixed by a redesign of the architeture that removes the deposit/withdraw functionalities from the `stSTRK` token.

6.6 [Info] Event emitted with incorrect parameter

File(s): src/contracts/stake_stark.cairo

Description: The `_delegator_withdraw(...)` is called to process withdrawals of the delegators which have already waited for `withdrawal_window_period`. After calling `process_withdrawal(...)` function on the delegator, `DelegatorStatusChanged` event is emitted which have two parameters, one for delegator address and other for available time which should be current timestamp. But rather than passing current timestamp, `0` is passed.

```
fn _delegator_withdraw(ref self: ContractState) {
    let now = get_block_timestamp();
    let mut i: u8 = 0;
    while i < self.num_delegators.read() {
        let (is_available, available_time) = self.delegator_status.read(i);

        // If the delegator is unavailable but the available time has passed
        if !is_available && now >= available_time {
            let delegator_address = self.delegators.read(i);
            let delegator = IDelegatorDispatcher { contract_address: delegator_address };

            let withdrawn_amount = delegator.process_withdrawal();
            self
                .emit(
                    Events::DelegatorWithdrew {
                        id: i, delegator: delegator_address, amount: withdrawn_amount
                    }
                );

            self.delegator_status.write(i, (true, now));
            // @audit available_time should be `now` and not `0`
            // as can be seen in the above line too.
            self
                .emit(
                    Events::DelegatorStatusChanged {
                        delegator: delegator_address, status: true, available_time: 0,
                    }
                );
        }

        i += 1;
    }
}
```

Recommendation(s): Replace `0` with `now` for emitting `DelegatorStatusChanged` event in `_delegator_withdraw(...)`.

Status: Fixed.

Update from the client: fixed. commit hash: 166fa43

6.7 [Info] Incorrect value assigned to BURNER_ROLE and OPERATOR_ROLE constant

File(s): src/utils/constants.cairo

Description: The value of roles is calculated using the following formula:

```
hex(int.from_bytes(Web3.keccak(text="ROLE_NAME"), "big") & mask_250)
```

The BURNER_ROLE and OPERATOR_ROLE have the following values assigned:

```
0x300b0bb5f166fb58587f4b1c6caed43a923bc0edcab7027c1a163433cc7dc3f // BURNER
0x23c157c0618fee210e9000399594099ceb3b2ce43c8f9e316ed8b04190307ad // OPERATOR
```

Whereas as per the formula, the values should be:

```
// hex(int.from_bytes(Web3.keccak(text="BURNER_ROLE"), "big") & mask_250)
0x11d16cbaffd01df69ce1c404f6340ee057498f5f00246190ea54220576a848 // BURNER
// hex(int.from_bytes(Web3.keccak(text="OPERATOR_ROLE"), "big") & mask_250)
0x3667070c54ef182b0f5858b034beac1b6f3089aa2d3188bb1e8929f4fa9b929 // OPERATOR
```

Recommendation(s): Consider correcting the values of BURNER_ROLE and OPERATOR_ROLE.

Status: Fixed.

Update from the client: fixed. hash: 1a042d9

6.8 [Info] Missing pause and unpause functions

File(s): src/contracts/delegator.cairo

Description: In Delegator contract, there is assert to make sure the contract isn't paused on all state changing functions except upgrade(...) but there isn't any function to pause or unpause the contract.

Recommendation(s): Add functions to pause and unpause in the Delegator contract.

Status: Fixed.

Update from the client: fixed. commit hash: 166fa43

6.9 [Info] Rewards distribution is not fully fair

File(s): src/contracts/stStark.cairo

Description: The stSTRK token increases its value over time with the accrual of rewards. The change in value for the token occurs when the rebase(...) function is executed, which happens every time rewards are accrued. As a result, all the stStark token holder benefits with a call to rebase(...).

stStark tokens are minted as a result of a deposit operation. A user can execute a deposit operation before a call to rebase(...) occurs, and it will receive part of the rewards accrued since the last rebase(...) operation, even if the newly deposited tokens were not used to generate these rewards. This distribution of rewards is not totally fair as it benefits users who did not contribute to the accrued rewards.

It is important to note that the impact of this issue can be mitigated by frequently calling rebase(...) operations.

Recommendation(s): Consider revisiting the reward distribution mechanism. Executing minting operations in an async way similar to what is described in the EIP 7540 could solve this problem.

Status: Acknowledged.

6.10 [Info] The PAUSER_ROLE isn't granted to any address

File(s): src/contracts/stStark.cairo

Description: The pause(...) and unpause(...) functions assert whether the caller has PAUSER_ROLE, whereas no address is granted the PAUSER_ROLE. Therefore, the contract cannot be paused without upgrading it.

Recommendation(s): Consider granting PAUSER_ROLE to an address responsible for pausing or unpausing the contract in the constructor.

Status: Fixed.

Update from the client: fixed. commit hash 166fa43

6.11 [Info] The `burn(...)` function doesn't assert with `BURNER_ROLE`

File(s): src/contracts/stStark.cairo

Description: The `burn(...)` function asserts whether the caller has `MINTER_ROLE`, whereas assert should be done for `BURNER_ROLE`. Since both roles are granted to the `StakeStark` contract, it will work as expected.

Recommendation(s): Consider replacing assert for `MINTER_ROLE` with `BURNER_ROLE` in the `burn(...)` function.

Status: Fixed.

Update from the client: fixed. hash: 4666c9d

6.12 [Info] The `delegate(...)` function uses an outdated delegation pool interface

File(s): src/contracts/delegator.cairo

Description: The `STRK` tokens users deposit are delegated to the Starknet Staking's Delegation Pool via the `delegate(...)` function in the `Delegator` contract. If that particular delegator is already a pool member, the tokens will be added to his existing delegation. Otherwise, the delegator must be registered as a member. This is done by calling the `enter_delegation_pool(...)` function on the `Pool` contract. The current version of the code expects that this function will return a `success` boolean value to assert that the pool was entered successfully. This, however, won't function properly since the interface of the `enter_delegation_pool(...)` function has been changed in the current implementation of the `Pool` contract, and it no longer returns a `boolean` value.

```
fn delegate(ref self: ContractState, amount: u256) {
    // ...
    if !self.is_in_pool.read() {
        // @audit-issue The enter_delegation_pool function does not return
        // boolean value.
        let success = pool
            .enter_delegation_pool(
                self.stake_stark.read(), amount_u128
            );
        assert(success, 'Failed to enter delegation pool');
        // ...
    } else {
        // @audit If the delegator is already a member,
        // then add to existing delegation.
        let new_total = pool
            .add_to_delegation_pool(
                get_contract_address(), amount_u128
            );
        assert(new_total > 0, 'Failed to add delegation pool');
    }
    // ...
}
```

As a result, the execution of the `delegate(...)` function would revert, and the delegation of `STRK` tokens wouldn't be possible.

Recommendation(s): Consider using the most up-to-date implementation of the Staking Contracts interfaces and monitoring them for updates before the actual deployment of the protocol.

Status: Fixed.

Update from the client: fixed. commit hash: 2d6b38f

6.13 [Info] The `get_withdrawable_amount(...)` and `get_all_withdrawal_requests(...)` functions have incorrect implementations

File(s): src/contracts/stake_stark.cairo

Description: The `get_withdrawable_amount(...)` and `get_all_withdrawal_requests(...)` function loops through the `withdrawal_requests` from `request_id` as `0`, and it breaks when `requested.assets == 0`. The issue is that it's possible that the initial few requests would have `assets` equal to `0`, whereas the `withdrawable` requests come after them, and breaking early would not take `withdrawable` requests into account.

Recommendation(s): Consider starting with the last `request_id`, similar to the implementation in the `get_available_withdrawal_requests(...)` function.

Status: Fixed.

Update from the client: fixed. hash: 131b6f0

Update from Nethermind: The issue is not currently fixed, the new implementations does not stop after finding the oldest valid request.

Update from the client: Fixed in commit hash 3e42e2.

6.14 [Best Practices] The `last_processing_time` and `last_reward_claim_time` state variables could be emitted as an event

File(s): `src/contracts/stake_stark.cairo`, `src/contracts/delegator.cairo`

Description: The `last_processing_time` state variable tracks the timestamp of the most recent batch processing. The `last_reward_claim_time` variable records the timestamp when rewards were last claimed. Both variables aren't tied to any logic in the codebase except for the view functions that allow off-chain components to query their values. If the purpose of the two variables is to inform the off-chain infrastructure about the state of the smart contract and make it react to it, then emitting this information in the form of an event might be more suitable.

Recommendation(s): To keep the code clean and readable, consider removing redundant state variables and emitting this information as an event. This way, the data needed for the on-chain part of the application is stored on-chain, while the off-chain part is connected with the help of the events.

Status: Fixed.

Update from the client: fixed. hash: b1f15da, 29845a4.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the Starkstake_ documentation

The documentation for the Starkstake_ was provided through their documentation. This documentation provided a high-level overview of the protocol and details of its implementation. Moreover, the Starkstake_ team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Compilation Output

```
scarb build
   Compiling snforge_scarb_plugin v0.31.0
   ↪  (git+https://github.com/foundry-rs/starknet-foundry.git?tag=v0.31.0#72ea785ca354e9e506de3e5d687da9fb2c1b3c67)
    Finished `release` profile [optimized] target(s) in 0.16s
   Compiling stakestark_ v0.1.0 (/Users/usr/Nethermind/NM-0337-StakeStark/Scarb.toml)
    Finished `dev` profile target(s) in 31 seconds
```

8.2 Tests Output

```
Collected 6 test(s) from starkstake_ package
Running 6 test(s) from src/
[PASS] starkstake_::contracts::tests::stark_stake_test::test_complex_batch_processing (gas: ~19419)
[PASS] starkstake_::contracts::tests::stark_stake_test::test_stark_stake_system_overall (gas: ~18894)
[PASS] starkstake_::contracts::tests::stark_stake_test::test_multiple_deposits_and_withdrawals (gas: ~18823)
[PASS] starkstake_::contracts::tests::integration_test::test_full_stake_unstake_cycle (gas: ~18297)
[PASS] starkstake_::contracts::tests::stark_stake_test::test_deposit_withdraw_cycle_with_rewards (gas: ~19614)
[PASS] starkstake_::contracts::tests::stark_stake_test::test_staggered_withdrawals (gas: ~19859)
Tests: 6 passed, 0 failed, 0 skipped, 0 ignored, 0 filtered out
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.