Security Review Report
NM-0397-WILDERWORLD

NETHERMIND SECURITY

(Mar 13, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for the smart contracts of WilderWorld. The audit scope comprises of smart contracts related to Staking, Voting, and DAO smart contracts.

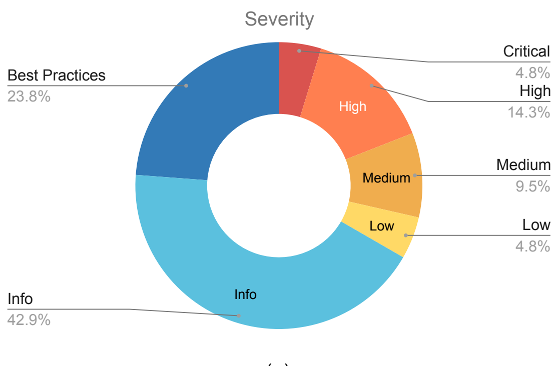
The staking contracts allow users to stake ERC20 or ERC721 tokens in locked or unlocked mode. In locked mode, the user's tokens are held for a fixed period, earning a higher yield. Conversely, unlocked mode offers more flexibility, allowing users to withdraw their funds at any time.

Upon staking, the protocol mints stake representation tokens, which serve as voting tokens for participation in ZDAO governance. The ZDAO governance is built based on OpenZeppelin's governance contracts.

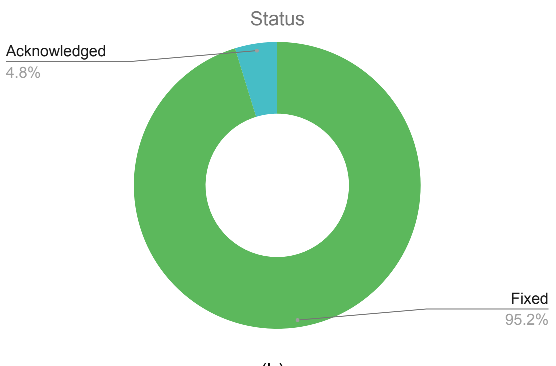
The audited code comprises 1093 lines of code written in the Solidity language. This security review focuses exclusively on the smart contracts listed in Section 2 (*Audited Files*).

The audit was performed using (a) manual analysis of the codebase and (b) creation of test cases. **Along this document, we report** 21 points of attention, where one is classified as `Critical`, three are classified as `High`, two are classified as `Medium`, one is classified as `Low`, and 14 are classified as `Informational` and `Best Practices`. 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.



(a)



(b)

Fig. 1: Distribution of issues: Critical (1), **High** (3), **Medium** (2), **Low** (1), **Undetermined** (0), **Informational** (9), **Best Practices** (5).
Distribution of status: Fixed (20), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jan 20, 2025
Response from Client	Regular responses during audit engagement
Final Report	Mar 13, 2025
Repository	zModules
Commit (Audit)	97e93a19797d3f40c63ec52b407b8caf2f14eb61
Commit (Final)	f7a82d285397326e5378ba17cf6b5d6b5f8d671b
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	voting/ZeroVotingERC20.sol	57	26	45.6%	10	93
2	voting/ZeroVotingERC721.sol	111	36	32.4%	26	173
3	voting/IZeroVotingERC20.sol	22	1	4.5%	6	29
4	voting/IZeroVotingERC721.sol	26	5	19.2%	13	44
5	dao/ZDAO.sol	189	48	25.4%	15	252
6	staking/StakingBase.sol	198	154	77.8%	55	407
7	staking/IStakingBase.sol	84	101	120.2%	37	222
8	staking/ERC20/IStakingERC20.sol	21	22	104.8%	12	55
9	staking/ERC20/StakingERC20.sol	149	85	57.0%	44	278
10	staking/ERC721/StakingERC721.sol	199	97	48.7%	50	346
11	staking/ERC721/IStakingERC721.sol	37	33	89.2%	17	87
	Total	**1093**	**608**	**55.6%**	**285**	**1986**

3 Summary of Issues

	Finding	Severity	Update
1	Unstaking with exit allows unauthorized rewards claim	Critical	Fixed
2	Potential reward manipulation due to incorrect reward multiplier	High	Fixed
3	Staking parameters change impact past rewards computations	High	Fixed
4	User loses rewards when unstaking partial amounts	High	Fixed
5	Array length manipulation allows partial exit in `_unstakeMany(...)`	Medium	Fixed
6	Automatic re-locking of unlocked tokens and rewards in staking flow	Medium	Fixed
7	Stale locked state might lead to reward discrepancies	Low	Fixed
8	Absence of reentrancy protection in staking contracts	Info	Fixed
9	Duplicated functions to receive native tokens	Info	Fixed
10	Incorrect setup of Multiplier for locked rewards could prevent staking	Info	Fixed
11	Incorrect total supply of tokens returned by `ZeroVotingERC721`	Info	Fixed
12	Unnecessary `payable` operator for unstaking functions	Info	Fixed
13	Unstaking non-locked tokens with exit cancels rewards	Info	Acknowledged
14	Unused `ERC20Permit` implementation in `ZeroVotingERC20`	Info	Fixed
15	Usage of `transfer()` to move native tokens	Info	Fixed
16	`msg.value` should be zero when staking ERC20 tokens	Info	Fixed
17	Errors declared in the interface but never used	Best Practices	Fixed
18	Missing zero address check for contract admin	Best Practices	Fixed
19	Redundant check for token ownership in `_unstakeMany`	Best Practices	Fixed
20	Redundant code when overriding the `_getContractRewardsBalance` function	Best Practices	Fixed
21	Unnecessary `stakers` mapping in `StakingERC721` contract	Best Practices	Fixed

4 System Overview

WilderWorld consists of a collection of modular smart contracts, organized into three core functional areas: Staking, Voting tokens and a DAO module. The following sections provide a detailed explanation of the interactions within these modules.

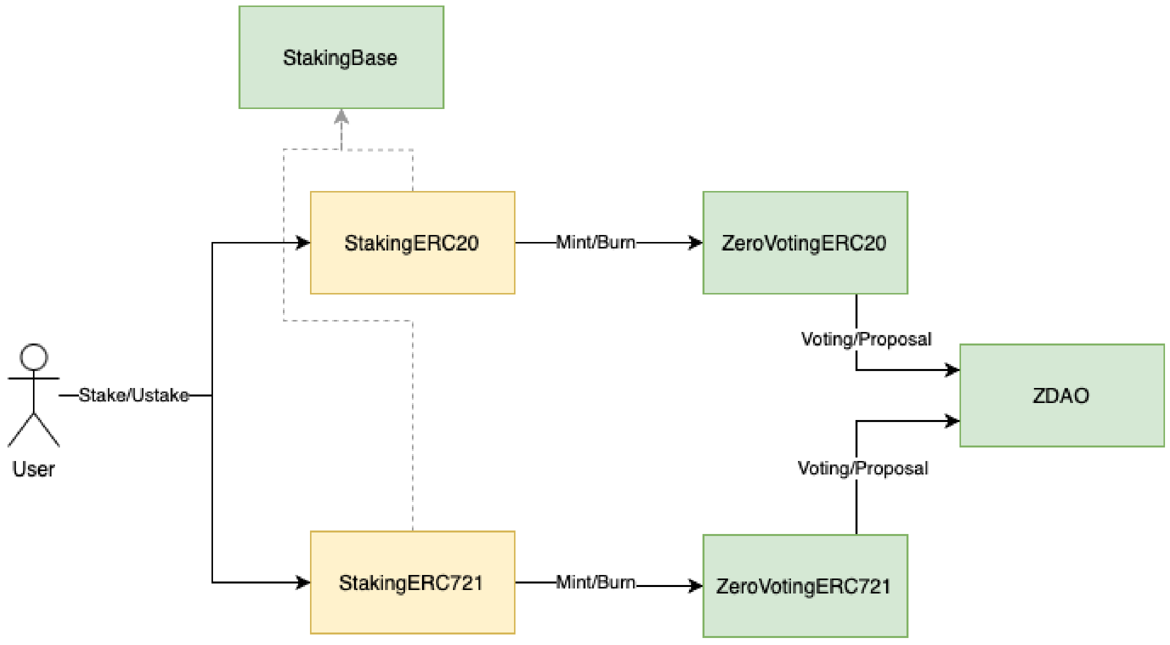


Fig. 2: WilderWorld overview

4.1 Staking

The staking module enables users to stake either ERC20 or ERC721 tokens through the `StakingERC20` and `StakingERC721` contracts, respectively. Upon staking, users receive a representative token that allows them to participate in the protocol's governance.

4.1.1 Staking via `StakingERC20`

Users can stake ERC20 or native tokens into the `StakingERC20` contract. Upon staking, an equivalent amount of a representative token is minted for the user. Staking can occur in two modes:

– Locked mode: In locked mode, the deposited tokens are locked for a defined period. The longer the lock, the higher the reward rate. Rewards are pre-computed and recorded at the time of staking. Users can stake with a lock using the `stakeWithLock(...)` function.

```
function stakeWithLock(uint256 amount, uint256 lockDuration) external payable override
```

– Non-locked mode: Unlocked mode allows users to stake tokens without locking them for a pre-defined period. Rewards are earned based on the duration between the deposit and unstake time. This is possible via the `stakeWithoutLock(...)` function.

```
function stakeWithoutLock(uint256 amount) external payable override
```

4.1.2 Staking via `StakingERC721`

The `StakingERC721` contract facilitates the staking of ERC721 tokens. Similar to ERC20 staking, users can stake with or without a lock. Upon staking an ERC721 token, a representative ERC721 token with the same tokenId is minted for the user.

– The `stakeWithLock(...)` function allows ERC721 staking with a lock.

```
function stakeWithLock(uint256[] calldata tokenIds, string[] calldata tokenUris, uint256 lockDuration) external
↪ override
```

– The `stakeWithoutLock(...)` function allows ERC721 staking without a lock.

```
function stakeWithoutLock(uint256[] calldata tokenIds, string[] calldata tokenUris) external override
```

4.1.3 Claiming Rewards

Rewards are calculated based on the amount staked, the staking duration, and the staking mode. The reward formula is:

```
rewards = rewardsMultiplier * amount * config.rewardsPerPeriod * timeDuration / config.periodLength / divisor
```

The accrued rewards are tracked in the contracts based on the mode of staking. While the rewards for unlocked mode are claimable at any time, the rewards for locked mode will become available only after the lock period expires. Users can claim their accrued rewards via the `claim(...)` function.

```
function claim() external override
```

4.1.4 Unstaking

Users can unstake their tokens by burning the representative token and receiving the staked amount. Rewards are calculated and transferred accordingly.

In the case of locked tokens, unstaking is only allowed after the lock period. However, if users wish to exit early, they can set the exit flag to true, which will force unstaking but invalidate accrued rewards. This behavior is consistent across both ERC20 and ERC721 staking. **Note that in the latest code, the exit functionality has been implemented in a separate exit(...) function.**

```
function exit(bool locked) external override nonReentrant

function exit(uint256[] memory _tokenIds, bool _locked) public override nonReentrant
```

For the `StakingERC20` contract, users can unstake non-locked funds using the `unstake(...)` function, renamed into `unstakeUnlocked(...)` in the latest version of the code.

```
function unstakeUnlocked(uint256 amount) external override
```

For locked funds, the `unstakeLocked(...)` function is used. This function reverts if funds are still within their lock period.

```
function unstakeLocked(uint256 amount) external payable override
```

For the `StakingERC721` contract, both locked and unlocked tokens can be unstaked using the `unstake(...)` function. This function was split in the latest code into two different functions similar to the `StakingERC20` contract:

```
function unstakeLocked(uint256[] memory _tokenIds) public override nonReentrant

function unstakeUnlocked(uint256[] memory _tokenIds) public override nonReentrant
```

4.2 Voting Module

The protocol supports voting for both ERC20 and ERC721 tokens. Users can delegate their voting rights to others, and voting tokens are non-transferable. The following contracts facilitate the voting functionality:

- The `ZeroVotingERC20` contract implements an ERC20 voting token.
- The `ZeroVotingERC721` contract implements an ERC721 voting token.

4.3 ZDAO

The protocol's ZDAO contract, built using OpenZeppelin governance modules, enables eligible users to create proposals, vote on them, and delegate votes to others. Proposals that meet the required threshold are executed and incorporated into the protocol's operations.

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] Unstaking with exit allows unauthorized rewards claim

File(s): StakingERC721.sol

Description: In the current implementation, when a user initiates an unstake operation with the `exit` flag set to `true` for a token that is still locked, the system behaves as expected: it unstakes the token but does not transfer rewards to the user. However, it fails to properly reset the `owedRewardsLocked` variable, which holds the pre-computed reward amount for the entire lock duration.

```
function _unstakeMany(...) internal {
    // ...
    for (i; i < _tokenIds.length;){
        // ...
        if (nftStaker.locked[_tokenIds[i]]) {
            // ...
             if (_getRemainingLockTime(nftStaker.stake) == 0) {
                  // ...
                  nftStaker.stake.owedRewardsLocked = 0; // @audit reset only when claimed
                   rewardsGivenLocked = true;
              }
              // ...
            _unstake(_tokenIds[i]);
            // ...
            }
            // ...
    }
  // ...
    if (!exit) {
            // Transfer the user's rewards
            _transferAmount(config.rewardsToken, rewards);
            emit Claimed(msg.sender, rewards);
    }
}
```

As a result, the user can claim these rewards via the `claim()` function in the case of a partial unstake. Alternatively, the user can make another stake to access these rewards. This allows malicious users to exploit the system by creating stakes with long lock durations, exiting the stake early, and then claiming pre-computed rewards, potentially depleting the contract's funds.

Recommendation(s): The `owedRewardsLocked` value should be reset when unstaking locked funds.

Status: Fixed

Update from the client: e327e43a434500450dcf80384d6217f06754349c

6.2 [High] Potential reward manipulation due to incorrect reward multiplier

File(s): StakingBase.sol

Description: In the _coreStake(...) function, when a user stakes a new amount with a locking period while their previous lock has not yet expired, the new stake is only locked for the remaining time of the previous lock. However, the rewards for the new stake are still calculated using the previous lock's rewardsMultiplier, which was intended for the entire previous lock duration. This results in the new stake receiving a disproportionately high rewards multiplier based on the full original lock period, even though it is only locked for the remaining time.

```
function _coreStake(...) internal {
    // ...
    if (block.timestamp > staker.unlockedTimestamp) {
        // ...
     } else {
         staker.owedRewardsLocked += _getStakeRewards(
                    amount,
                    staker.rewardsMultiplier, //@audit multiplier for the previous lock duration
                    _getRemainingLockTime(staker),
                    true);
    }
}
```

This issue can be exploited by users who stake large amounts near the end of their previous lock period. They can benefit from a higher rewards multiplier based on the full lock duration despite their new stake being locked for a much shorter period.

Recommendation(s): The rewardsMultiplier should be recomputed to reflect the remaining lock time for the new stake rather than relying on the multiplier from the previous lock.

Status: Fixed

Update from the client: e7ec0dab2c5b6b8bf14adc562d4748277aaad641

6.3 [High] Staking parameters change impact past rewards computations

File(s): StakingBase.sol

Description: The owner of the staking contracts has the ability to modify configuration values that impact reward calculations, notably:

```
uint256 rewardsPerPeriod
uint256 periodLength
uint256 minimumRewardsMultiplier
uint256 maximumRewardsMultiplier
```

However, in the current design, changes to these parameters impact rewards computation for historical stakes and not the newest ones only.

For example, if a user stakes tokens and the owner later changes the reward parameters during the staking period (updates the rewardsPerPeriod, for example), the system currently recalculates all previous rewards using the new configuration. This leads to a jump in accrued rewards, where rewards from earlier periods are either over- or under-calculated, depending on the direction of the change.

```
function _getStakeRewards(...) internal view returns (uint256) {
    uint256 divisor = locked ? LOCKED_PRECISION_DIVISOR : PRECISION_DIVISOR;

    return rewardsMultiplier * amount * config.rewardsPerPeriod * timeDuration / config.periodLength / divisor;
}
```

Recommendation(s): Ensure that changes to configuration values should only apply to rewards from the time they are updated and should not affect rewards for past periods. Alternatively, all user rewards must be accounted for and stored before performing any parameter update.

Status: Fixed

Update from the client: dd2a9ecfd5bd78d368d89fcb30609171b20356a2

6.4 [High] User loses rewards when unstaking partial amounts

File(s): StakingERC20.sol, StakingERC721.sol

Description: When a user unstakes a portion of their tokens, the contract computes rewards based on the withdrawn amount rather than the total amount staked. Refer to the code snippet below, where accrued rewards are being computed using the amount withdrawn.

```
function _unstake(uint256 amount, bool locked, bool exit) internal {
    // ...
    // most recent timestamp?
    rewards = staker.owedRewards + _getStakeRewards(
        amount, //@audit only rewards for unstaked amount are taken
        1, // Rewards multiplier
        block.timestamp - staker.lastTimestamp,
        false
    );
    if (staker.amountStaked == amount) {
       ...
    } else {
        staker.amountStaked -= amount;
        staker.lastTimestamp = block.timestamp;
    }
    // ...
}
```

After the partial unstake, the contract updates the staker.lastTimestamp to the current block.timestamp, considering that the entire staking period rewards are accounted for. Consequently, users lose the accrued rewards for the remaining staked amount.

Note that the present issue applies to both the ERC20 and ERC721 staking contracts, including both locked and non-locked tokens.

Recommendation(s): Use the total staked amount instead of the withdrawn amount when calculating the accrued rewards. For ERC20Staking, the staker.amountStaked can be used, while in ERC721Staking, the number of staked tokens must be accounted for.

Status: Fixed

Update from the client: 96191f24125a69ec457ccb73f6ed3457e8dda98d

6.5 [Medium] Array length manipulation allows partial exit in _unstakeMany(...)

File(s): StakingERC721.sol

Description: The staking process is designed to enforce that a user must perform a full exit when unstaking. In the StakingERC721 case, this is enforced by the following check in the _unstakeMany(...) function.

```
if (exit && _tokenIds.length < nftStaker.stake.amountStakedLocked) {
        revert NotFullExit();
}
```

However, the function currently allows users to submit arbitrary or invalid token IDs. These invalid IDs are skipped during processing, but they are still counted in the array length. As a result, users can manipulate the array length to bypass the full exit check and perform a partial exit.

```
if (
    nftStaker.staked[_tokenIds[i]] == false
    || IERC721(config.stakeRepToken).ownerOf(_tokenIds[i]) == address(0)
    || IERC721(config.stakeRepToken).ownerOf(_tokenIds[i]) != msg.sender
) {
    // Either the list of tokenIds contains a non-existent token
    // or it contains a token the owner doesnt own
    unchecked {
        ++i;
    }
    continue;
}
```

Recommendation(s): Update the function logic to avoid relying on the array length, as it can be manipulated.

Status: Fixed

Update from the client: b18e44ff7b34803973f19508f5b273d1f6c3b169

6.6 [Medium] Automatic re-locking of unlocked tokens and rewards in staking flow

File(s): `StakingBase.sol`

Description: Currently, when an existing user stakes additional tokens in locked mode and the previous lock duration has expired, the system automatically includes the previously unlocked amount as part of the new staked and locked amount.

From the user's perspective, the initially locked tokens are unlocked, and their rewards become claimable. However, if the user stakes additional tokens, the previously unlocked tokens are re-locked for a new period, alongside the new tokens. As a result, the previously accumulated rewards are also locked again. If the user attempts to exit during this new lock period, they risk losing the rewards earned during the previous lock period, even though the original lock was successfully completed.

```
function _coreStake(
    Staker storage staker,
    uint256 amount,
    uint256 lockDuration
) internal {
    if (lockDuration == 0) {
        // ...
    } else {
        if (block.timestamp > staker.unlockedTimestamp) {
            // The user has never locked or they have and we are past their lock period

            // get the user's owed rewards from period in between `unlockedTimestamp` and present at rate of 1
            uint256 mostRecentTimestamp = _mostRecentTimestamp(staker);

            // this will return 0 if `amountStakedLocked` == 0
            staker.owedRewardsLocked += _getStakeRewards(
                staker.amountStakedLocked,
                1,
                block.timestamp - mostRecentTimestamp,
                false
            );

            // Then we update appropriately
            staker.unlockedTimestamp = block.timestamp + lockDuration;
            staker.rewardsMultiplier = _calcRewardsMultiplier(lockDuration);

            // We precalculate the amount because we know the time frame
            staker.owedRewardsLocked += _getStakeRewards(
                amount,
                staker.rewardsMultiplier,
                lockDuration,
                true
            );
        } else {...}

        staker.lastTimestampLocked = block.timestamp;
        //@audit the cumulative amount is locked
        staker.amountStakedLocked += amount;
    }
}
```

Recommendation(s): Consider allowing users to unstake and claim their locked rewards once the timelock has expired. Alternatively, document the current process clearly to avoid user mistakes that could result in the locking or loss of accrued rewards.

Status: Fixed

Update from the client: f9201fdd4b601dc519b61a4e5591e8806d61f295

6.7 [Low] Stale locked state might lead to reward discrepancies

File(s): StakingERC721.sol

Description: The nftStaker.locked field is set during staking and stores whether a token Id is locked or not.

During the unstaking process of a locked token, this field is not reset to false.

```
function _unstakeMany(...) internal {
    // ...
    for (i; i < _tokenIds.length;){
        // ...
        if (nftStaker.locked[_tokenIds[i]]) {
            // ...
            _unstake(_tokenIds[i]);
            --nftStaker.stake.amountStakedLocked;
            nftStaker.staked[_tokenIds[i]] = false;
            isAction = true;
            }
            // ...
    }
  // ...
}
```

As a result, if the same token is restaked without a lock, the stale locked entry remains true. This creates an inconsistency where the token is marked as locked in the system, but no rewards are actually computed for the locked stake.

Recommendation(s): Ensure that the nftStaker.locked flag is properly reset to false after a token is unstaked.

Status: Fixed

Update from the client: 53967b6575a736168e944fef0c2fa3ef1ce3d25d

6.8 [Info] Absence of reentrancy protection in staking contracts

File(s): StakingBase.sol

Description: The StakingBase contract inherits from OpenZeppelin's ReentrancyGuard to prevent reentrancy vulnerabilities. However, for this protection to work, functions within the StakingBase contract and its derived contracts must be explicitly marked with the nonReentrant modifier.

Currently, the nonReentrant modifier has not been applied to any functions in the StakingBase contract nor in its derived contracts (StakingERC20 and StakingERC721).

Recommendation(s): Assign the nonReentrant modifier to the necessary staking functions.

Status: Fixed

Update from the client: 889de3387450e53049edd8002a72f0d701d66bc6

6.9 [Info] Duplicated functions to receive native tokens

File(s): StakingBase.sol

Description: In the current implementation, both the receive() and fallback() functions are defined, but both have empty logic. The receive() function is designed to handle native token transfers without data, while the fallback() function can handle transfers with or without data.

Since both functions serve the same purpose — allowing the contract to receive native tokens — and neither contains any additional logic, only one function is necessary.

Recommendation(s): Remove the redundant fallback() function.

Status: Fixed

Update from the client: Fixed in commits: e9474a1506a4c0fd0d02b7189a7de64e16c9df80 and 491a40af6e4c404fc7c33cb9f92ea9ea4db06caf.

6.10 [Info] Incorrect setup of Multiplier for locked rewards could prevent staking

File(s): StakingBase.sol

Description: The locked rewards are calculated based on reward multipliers and the duration for which the tokens are locked. One of the factors for lock rewards calculation is the difference between maximumRewardsMultiplier and minimumRewardsMultiplier configured by the owner of the staking contracts.

```
function _calcRewardsMultiplier(uint256 lock) internal view returns (uint256) {
    return config.minimumRewardsMultiplier
    + (config.maximumRewardsMultiplier - config.minimumRewardsMultiplier)
    * (lock)
    / config.periodLength;
}
```

However, the setter functions for these parameters lack validation to ensure that maximumRewardsMultiplier is already greater than minimumRewardsMultiplier. Setting invalid values will prevent users from staking tokens for a locked duration. That is because when the _calcRewardsMultiplier(...) function is called during staking, an underflow exception will occur.

Likewise, the function reverts when the periodLength is set to 0.

Recommendation(s): Add validations in setMinimumRewardsMultiplier and setMaximumRewardsMultiplier setter functions to enforce that maximumRewardsMultiplier is always greater than minimumRewardsMultiplier.

Additionally, ensure that periodLength is always greater than 0 to avoid potential reverts.

Status: Fixed

Update from the client: 5436a2a9b2ea2e5d924f21c61d04409041841d73

6.11 [Info] Incorrect total supply of tokens returned by ZeroVotingERC721

File(s): ZeroVotingERC721.sol

Description: The total supply of tokens in ZeroVotingERC721 contract is tracked using the _totalSupply state variable. The total supply should reflect the number of tokens in circulation after excluding the tokens that were burnt.

The current implementation of ZeroVotingERC721 contract increments the _totalSupply state variable when minting the new tokens through mint(...) and safeMint(...) functions, but it is not decremented when tokens are burnt through burn(...) function.

Therefore, _totalSupply does not correctly reflect the number of tokens currently in circulation.

Recommendation(s): The _totalSupply state variable should be decremented when a token is burnt to reflect the total number of tokens in circulation correctly.

Status: Fixed

Update from the client: 5f0a8b923e20eb3a531031a612cbe68c1d7fe27a4f06993d9

6.12 [Info] Unnecessary payable operator for unstaking functions

File(s): StakingERC20.sol

Description: The payable operator is unnecessary for unstaking functions in StakingERC20 contract. These functions do not require any native token payment.

```
function unstake(uint256 amount, bool exit) external payable override {
    _unstake(amount, false, exit);
}

function unstakeLocked(uint256 amount, bool exit) external payable override {
    _unstake(amount, true, exit);
}
```

Recommendation(s): Remove the payable operator from the above functions.

Status: Fixed

Update from the client: 8bd96366c7b2963bbfc4786b05a6017d42e07c07

6.13 [Info] Unstaking non-locked tokens with exit cancels rewards

File(s): StakingERC20.sol, StakingERC721.sol

Description: The staking contract allows users to stake tokens in both locked and unlocked modes. In locked mode, users earn higher rewards, but the staked tokens cannot be withdrawn before the end of the lock period. If a user attempts to exit from a locked stake, they forfeit all rewards. However, in unlocked mode, users have the flexibility to withdraw their stake at any time without forfeiting their rewards.

The current implementation of the `StakingERC20` contract incorrectly implements this logic. When unstaking, if the user sets the `exit` flag to `true`, rewards are lost even in non-locked mode.

```
function _unstake(uint256 amount, bool locked, bool exit) internal {
    // ...
    if (locked) {...}
    else { // @audit non-locked mode
        if (amount > staker.amountStaked) {
            revert UnstakeMoreThanStake();
        }

        if (exit) {
            rewards = 0; //@audit rewards are removed
            staker.owedRewards;
        } else {...}
        // ...
    }
    // ...
}
```

This issue is also present in the unstaking of ERC721 tokens.

Recommendation(s): Revise the logic to stop forfeiting rewards in unlocked mode. The `exit` flag should be applicable only to the locked mode of staking.

Status: Acknowledged

Update from the client:

Update from Nethermind Security: The Wilderworld team confirmed during a call that this behavior is intentional. The exit flow has been restructured as a separate function, allowing users to withdraw only their initially staked tokens. This new functionality is particularly useful in cases where the contract is unable to provide reward tokens.

6.14 [Info] Unused `ERC20Permit` implementation in `ZeroVotingERC20`

File(s): ZeroVotingERC20.sol

Description: The `ZeroVotingERC20` contract inherits from the `ERC20Permit` contract but restricts token transfers through the `_update()` function, which is intended to prevent transfers of tokens. The `ERC20Permit` implementation introduces the `permit(...)` function that allows users to approve token transfers using signed messages. However, since the `ZeroVotingERC20` contract explicitly disables token transfers, the permit(...) function becomes non-functional and, therefore, inheriting from the `ERC20Permit` contract is redundant.

```
function _update(
    address from,
    address to,
    uint256 value
) internal override(ERC20, ERC20Votes) {
    if (from != address(0) && to != address(0)) {
    revert NonTransferrableToken();
    }
    super._update(from, to, value);
}
```

Recommendation(s): Remove the unused inheritance of `ERC20Permit` from the `ZeroVotingERC20` contract.

Status: Fixed

Update from the client: 7f7c8bf0fc4c5a5016fca89a3b4d09c21e744331

6.15 [Info] Usage of `transfer()` to move native tokens

File(s): `StakingBase.sol`

Description: In order to transfer native tokens, the `StakingBase` contract currently uses the `transfer` function. This is not a recommended function to move native tokens due to a fixed gas limit of 2300. As the gas cost is subject to change as part of infra upgrades.

Additionally, when the recipient is a smart contract like a multi-sig wallet, 2300 gas might not cover the gas required to complete the transfer, leading to a revert.

Recommendation(s): Use a low-level `.call` function to move native tokens. Additionally, the reentrancy guard should be implemented in different functionalities to prevent reentrancy attacks on both `ERC20Staking` and `ERC721Staking` contracts.

Status: Fixed

Update from the client: 2b8c8a51ba4de33b920901195873f0ad2a792ee2

6.16 [Info] `msg.value` should be zero when staking ERC20 tokens

File(s): `StakingERC20.sol`

Description: In the `StakingERC20` contract, the staking token can either be an ERC20 token or the native gas token.

For gas tokens, funds are received through the payable operator of the `stake(...)` function. However, when the staking token is an ERC20 token, the function does not enforce `msg.value` to be zero. As a result, users may inadvertently send `msg.value` alongside the ERC20 token amount, potentially leading to the loss of funds.

```
function _stake(uint256 amount, uint256 lockDuration) internal {
    if (amount == 0) {
        revert ZeroValue();
    }

    // @audit msg.value must be 0 when stakingToken !=address(0)
    if (config.stakingToken == address(0)) {
        if (msg.value != amount) {
            revert InsufficientValue();
        }
    }

    Staker storage staker = stakers[msg.sender];
```

Recommendation(s): Add a check to ensure `msg.value` is 0 when `config.stakingToken != address(0)`

Status: Fixed

Update from the client: 320f3331259c610def9aa55e2a38fb208734e634

6.17 [Best Practices] Errors declared in the interface but never used

File(s): `IStakingBase.sol`

Description: The `IStakingBase` interface defines the following errors, but they are not used in any of the implementation contracts:

```
error InvalidStake()
error CannotClaim()
```

Recommendation(s): Remove the unused errors declaration.

Status: Fixed

Update from the client: 152b9e717b3bed7f40c7d49120cacb536f9644de

6.18 [Best Practices] Missing zero address check for contract admin

File(s): ZeroVotingERC20.sol

Description: The ZeroVotingERC20 contract's constructor takes the admin address as an input. This address is granted the DEFAULT_-ADMIN_ROLE, BURNER_ROLE, and MINTER_ROLE roles within the contract. However, the function does not check that this address is not set to address(0).

```
constructor(
        string memory name,
        string memory symbol,
        address admin
    ) ERC20(name, symbol)  ERC20Permit(name){
        //@audit Missing zero address check on admin
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(BURNER_ROLE, admin);
        _grantRole(MINTER_ROLE, admin);
}
```

Recommendation(s): Implement a check in the constructor, ensuring admin cannot be set to address(0).

Status: Fixed

Update from the client: 80e57e732a7d920f97469d97c5154b2b0adb1ac2

6.19 [Best Practices] Redundant check for token ownership in _unstakeMany

File(s): StakingERC721.sol

Description: The _unstakeMany(...) function processes an array of token IDs to be unstaked. For each token, the function checks whether the caller owns it and whether it was previously staked in the contract. This is done through the following checks:

```
if (
      nftStaker.staked[_tokenIds[i]] == false
      || IERC721(config.stakeRepToken).ownerOf(_tokenIds[i]) == address(0) //@audit Unnecessary check
      || IERC721(config.stakeRepToken).ownerOf(_tokenIds[i]) != msg.sender
  ) {
        // Either the list of tokenIds contains a non-existent token
        // or it contains a token the owner doesnt own
        unchecked {
            ++i;
        }
        continue;
}
```

However, the second check is redundant. Specifically, verifying that the token's owner is not address(0) is unnecessary, as the subsequent check already covers this condition, ensuring the token owner is msg.sender.

Recommendation(s): Remove the redundant check from the code above.

Status: Fixed

Update from the client: 5d72252f912461d04f115f2fb61c39ec660881ab

6.20 [Best Practices] Redundant code when overriding the `_getContractRewardsBalance` function

File(s): `StakingERC20.sol`

Description: The `StakingERC20` contract inherits from `StakingBase` and overrides the `_getContractRewardsBalance()` function to account for staked tokens when calculating the total rewards in the contract. However, the first part of the function duplicates logic from the base contract's overridden function.

To eliminate redundancy, this code can be replaced by calling `super._getContractRewardsBalance()` instead.

```
function _getContractRewardsBalance() internal view override returns (uint256) {
    uint256 balance;
    //@audit Following if-else block can be replaced by a call to `super._getContractRewardsBalance()`
    if (address(config.rewardsToken) == address(0)) {
        balance = address(this).balance;
    } else {
        balance = IERC20(config.rewardsToken).balanceOf(address(this));
    }

    if (config.rewardsToken == config.stakingToken) {
        return balance - totalStaked;
    }

    return balance;
}
```

Recommendation(s): Replace the redundant code in `_getContractRewardsBalance()` with a call to `super._getContractRewardsBalance()`.

Status: Fixed

Update from the client: 06d92d64ae2689eaba672b31b6f30a051fd9e20a

6.21 [Best Practices] Unnecessary `stakers` mapping in `StakingERC721` contract

File(s): `StakingBase.sol` `StakingERC721.sol`

Description: The `StakingBase` contract implements common functionalities for staking both ERC20 and ERC721 tokens. It includes the `stakers` mapping, which is used to track each staker's data via the `Staker` struct.

```
contract StakingBase is Ownable, ReentrancyGuard, IStakingBase {
    ...
    /**
     * @notice Mapping of each staker to that staker's data in the `Staker` struct
     */
    mapping(address user => Staker staker) public stakers;
    ...
}
```

However, the `stakers` mapping is only used in the `StakingERC20` contract. Since the mapping is declared in the base contract, it becomes part of the storage layout for the `StakingERC721` contract as well, which results in redundancy and unnecessary storage usage.

Recommendation(s): Consider moving the `stakers` mapping to `StakingERC20` interface and implementation contract.

Status: Fixed

Update from the client: 0310dcd7abf48166fd9ee4b847066b74db055d73

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 Wilderworld documentation

The `Wilderworld` team was actively present in regular calls, effectively addressing concerns and questions raised by the Nethermind Security team. Additionally, the code includes NatSpec documentation for different functions and their parameters. However, the project documentation could be improved by providing a more comprehensive written overview of the system and an explanation of the different design choices.

8 Test Suite Evaluation

8.1 Compilation Output

```
> yarn build
yarn run v1.22.22
$ yarn run clean && yarn run compile
$ hardhat clean
WARNING: You are currently using Node.js v19.9.0, which is not supported by Hardhat. This can lead to unexpected
 ↪ behavior. See https://hardhat.org/nodejs-versions


$ hardhat compile
WARNING: You are currently using Node.js v19.9.0, which is not supported by Hardhat. This can lead to unexpected
 ↪ behavior. See https://hardhat.org/nodejs-versions


Generating typings for: 80 artifacts in dir: typechain for target: ethers-v6
Successfully generated 220 typings!
Compiled 80 Solidity files successfully (evm target: paris).
$ yarn save-tag
$ chmod a+x ./src/utils/git-tag/save-tag.sh && bash ./src/utils/git-tag/save-tag.sh
Last tag: v0.8.0
Commit: e5b66adeb7fcb643ef8c8e2391ebde1ec0b3434d
Git tag saved to ./artifacts/git-tag.txt
Done in 17.17s.
```

8.2 Tests Output

```
> yarn test
yarn run v1.22.22
$ hardhat test
WARNING: You are currently using Node.js v19.9.0, which is not supported by Hardhat. This can lead to unexpected
 ↪  behavior. See https://hardhat.org/nodejs-versions
  Escrow Contract
     Should properly support deflationary a token
    Deploy
       Should set the right owner
       Should set the right token contract
       Should assign operators in the constructor if they are passed
       Should revert if _token is passed as a non-contract address
    Fund Management
       Should allow deposits
       Should re-deposit
       Should allow withdrawal
       Should handle withdrawal after multiple deposits
       Should revert on attempt to withdraw with zero balance
       Should revert if depositing zero amount
       Should revert if withdrawing zero amount
       Should revert withdrawals when balance is zero or withdrawing more than balance
       Should handle multiple consecutive deposits and withdrawals correctly
    Deposits of different grade amounts
       Should handle deposit of 0 wei correctly
       Should handle deposit of 1 wei correctly
       Should handle deposit of 100 wei correctly
       Should handle deposit of 10000000 wei correctly
       Should handle deposit of 1000000000000000000 wei correctly

  Governor Voting Flow Test
     Should properly retrieve and count votes during voting
     Should properly retrieve and count votes during voting (ERC721)

  Match Contract
     Should #setGameFeePercentage() correctly
     Should NOT allow operator to call #setGameFeePercentage()
     Should revert if feeVault is passed as 0x0 address
    Aux Operations
       #setFeeVault should set the address correctly and emit an event
       #setFeeVault() should revert if called by non-owner
       #setFeeVault() should revert if 0x0 address is passed
    Matches
      #startMatch()
         Should revert if called by a non-owner/non-operator
         Should fail if any player is not funded
         Should not start a match with an empty players array
         Should start match for all funded players and emit MatchStarted event with correct parameters
         Should exempt the entry fee from all the player balances
         Should save and lock the correct amount of fees for the match
         Should fail if the match already exists
         Should fail when starting a match with 0 `matchFee`
      #endMatch()
         Should revert if called by a non-owner/non-operator
         Should fail if the match does not exist
         Should fail if players and payouts array lengths mismatch
         Should revert if payout amounts are calculated incorrectly
         Should end the match and emit event with correct parameters
         Should remove the locked amount from the #lockedFunds mapping
         Should disperse the payouts correctly to the player #balances
         Should add #gameFee to the #feeVault balance
         Players should be able to withdraw their winnings
    Access Control
       owner, operators assigned at deploy and operators assigned later should have appropriate rights
    Deploy
       Deployed contract should exist in the DB
       Should be deployed with correct args
       Should have correct db and contract versions (49ms)
    Separate tokens
       Should deploy contract with mock, provided separetely from campaign
```

```
OwnableOperable Contract
  Ownable
     should assign msg.sender as owner
     should #transferOwnership() if owner is the caller
     should revert when #transferOwnership() is called by non-owner
  Operable
     should assign an operator if owner is the caller and emit an event
     should revert when #addOperator() is called by non-owner
     should revert when #addOperators() is called by non-owner
     should revert it zero address is passed as operator
     #addOperator() should revert if adding an existing operator
     #removeOperator() should revert if removing a non-existing operator
     should remove an operator if owner is the caller and emit an event
     should revert when #removeOperator() is called by non-owner
     should support multiple operators

StakingBase Unit Tests
  State Setters & Getters
     #setRewardsPerPeriod() should set value correctly
     #setRewardsPerPeriod() should revert if called by non-owner
     #setPeriodLength() should set value correctly
     #setPeriodLength() should revert if called by non-owner
     #setMinimumLockTime() should set value correctly
     #setMinimumLockTime() should revert if called by non-owner
     #setMinimumRewardsMultiplier() should set value correctly
     #setMinimumRewardsMultiplier() should revert if called by non-owner
     #setMaximumRewardsMultiplier() should set value correctly
     #setMaximumRewardsMultiplier() should revert if called by non-owner
     #getStakingToken() should return correct value
     #getRewardsToken() should return correct value
     #getStakeRepToken() should return correct value

StakingERC20
  #getContractRewardsBalance
     it accounts for balance when rewards and stake are same token
     Allows a user to see the total rewards remaining in a pool
  #stake
     Can stake without a lock successfully and mint proper amount of `stakeRepToken`
     Can stake a second time without a lock as the same user successfully
     Can stake with a lock successfully and mint proper amount of `stakeRepToken`
     Can stake a second time with a lock as the same user successfully and get proper amount of `stakeRepToken`
     Calculates in between rewards correctly after initial lock duration is complete
     Does not update the amount of remaining time on follow up stakes with lock
     Can stake as a new user without lock when others are already staked
     Can stake as a new user with a lock when others are already staked
     Fails when the staker locks for less than the minimum lock time
     Fails when the staker doesn't have the funds to stake
     Fails when the staker tries to stake 0
  #getRemainingLockTime
     Allows the user to view the remaining time lock period for a stake
     Returns 0 for a user that's passed their lock time
     Returns 0 for a user that has not staked
  #getPendingRewards
     Allows the user to view the pending rewards for a stake without a lock
     Updates their pending rewards appropriately when staking again without a lock
     Returns 0 for a user that is staked with a lock that they have not passed
     Includes the locked rewards in the sum when a user has passed their lock period
     Returns 0 for a user that has not staked
  #claim
     Allows the user to claim their non-locked rewards
     Allows the user to claim their locked rewards only when passed their lock
     Fails when the user has never staked
     Fails when the contract has no rewards
     Fails to claim when the user has not passed their lock time
 #unstake
     Allows a user to unstake non-locked amount partially and burns `stakeRepToken` (40ms)
     Allows a user to fully withdraw their entire non-locked staked amount and burns `stakeRepToken`
     Fails to unstake 0 amount
     Fails when the user has never staked
     Fails when the user tries to unstake more than they have staked
     Fails when there are not enough rewards in the contract
```

```
    #unstakeLocked
      Allows a user to partially unstake locked funds when passed their lock time and burns `stakeRepToken`
      Allows a user to fully unstake locked funds when passed their lock time and burns `stakeRepToken`
      Fails when the user tries to unstake 0 amount
      Fails when the user has never staked
      Fails when the user has not passed their lock time
      Fails when the user tries to unstake more than they have staked
      Fails when there are not enough rewards in the contract
    #unstakeLocked with 'exit'
      Allows a user to fully unstake using 'exit' within lock duration and burns `stakeRepToken`
      Doesn't effect non-locked funds when user fully unstakes using 'exit' after lock duration
      Allows the user to unstake with `exit` for non-locked funds
      Fails when the user tries to exit with less than their full amount staked
      Fails when the user tries to unstake 0 amount with `exit`
      Fails when the user has never staked
      Fails when the user tries to unstake more than they have staked
      Does not fail when there are not enough rewards in the contract
    #withdrawLeftoverRewards
      Allows the admin to withdraw leftover rewards
      Fails when the caller is not the admin
      Fails when the contract has no rewards left to withdraw
    #getStakerData
      Allows a user to get their staking data
      Returns zero values for a user that has not staked
    #getTotalPendingRewards
      Allows the user to view the total pending rewards when passed lock time
      Returns 0 for a user that has not staked
    Utility functions
      Finds the minimum lock time required to exceed non-locked rewards
      Tries to claim when RM is minimal value
    Other configs
      Stakes, claims, partially and fully unstakes when stake and reward token are chain token
      Fails when using native token and `amount` does not equal `msg.value`
      Stakes, claims, and unstakes correctly with an entirely different config (39ms)
    Events
      Emits a Staked event when a user stakes without a lock
      Emits a Staked event when a user stakes with a lock
      Emits a Claimed event when a user claims rewards
      Emits Unstaked and Claimed event when a user unstakes non-locked funds
      Emits an Unstaked event when unstaking locked funds passed the lock period
      Emits an Unstaked event on locked funds when user calls with `exit` as true
      Emits 'LeftoverRewardsWithdrawn' event when the admin withdraws

  StakingERC721
    - calcs correctly
     Should NOT deploy with zero values passed
    #getContractRewardsBalance
      Allows a user to see the total rewards remaining in a pool
    #stake
      Can stake a single NFT using #stakeWithoutLock and mint `stakeRepToken`
      Can stake a single NFT using #stakeWithLock
      Can stake multiple NFTs using #stakeWithoutLock
      Can stake multiple NFTs using #stakeWithLock
      Fails to stake when the token is already staked
      Does not modify the lock duration upon follow up stake (48ms)
      Does not modify the lock duration when a follow up stake provides a smaller lock duration (50ms)
      Fails to stake when the token id is invalid
      Fails to stake when the caller is not the owner of the NFT (54ms)
    #getRemainingLockTime
      Allows the user to view the remaining time lock duration (50ms)
      Returns 0 for users that did not stake the given token
      Returns 0 for a user that is past their lock duration
    #getPendingRewards
      Can view pending rewards for a user (54ms)
      Returns 0 for users that have not passed any time
      Returns 0 for users that have not staked
    #claim
      Can claim rewards when staked without time lock (54ms)
      Fails to claim when only locked stake and not enough time has passed (50ms)
      Claims when caller has both locked and unlocked stake even if the lock duration is not complete (50ms)
      Claims full rewards when both locked and not locked stakes exist and the lock duration has passed (50ms)
      Fails to claim when the caller has no stakes
    #unstake
      Can unstake a token that is not locked (57ms)
      Can unstake multiple unlocked tokens (59ms)
      Fails to unstake locked tokens without 'exit' when not enough time has passed (49ms)
      Fails to unstake when token id is invalid
      Fails to unstake when caller is not the owner of the SNFT
      Fails to unstake when token id is not staked
    #unstake with 'exit'
      Fails if the caller does not own the sNFT
      Fails if the sNFT is invalid
      Allows the user to remove their stake within the timelock period without rewards (51ms)
      Allows the user to remove multiple stakes within the timelock period without rewards
    Events
      Staking emits a 'Staked' event (45ms)
      Emits Staked event when calling with lock
      Staking multiple tokens emits multiple 'Staked' events
      Emits multiple staked events when calling with lock for multiple tokens
      Claim emits a 'Claimed' event (48ms)
      Unstake Emits 'Unstaked' and 'Claimed 'events (51ms)
    Other configs
      Can set the rewards token as native token
      Can set the rewards token as native token when locking (78ms)
      Can't use the StakingERC721 contract when an IERC20 is the staking token
      Can't use the StakingERC721 contract when an IERC721 is the rewards token (5576ms)
      Can't transfer rewards when there is no rewards balance (57ms)
    Utility functions + ZeroVotingERC721 standard functions
Locked rewards:    10000000000000000000000
Unlocked rewards:  1000000000000000000000
      Calculates the users rewards when they lock based on their lock time (48ms)
      #setBaseURI() should set the base URI (48ms)
      #setTokenURI() should set the token URI and return it properly when baseURI is empty
     #stake() with passed tokenURI should set the token URI when baseURI is empty and change back to baseURI when needed
      #withdrawLeftoverRewards() should withdraw all remaining rewards and emit an event
      #withdrawLeftoverRewards() should revert if contract balance is 0
      #withdrawLeftoverRewards() should only be callable by the owner
      #supportsInterface() should return true for ERC721 interface or interface of staking contract
      Should allow to change ownership
      should allow to renounce ownership

  Voting tokens tests
    ZeroVotingERC20
      Should correctly set name and symbol for ERC20 token
      tokens should NOT be transferrable
     Voting functions
        Should delegate votes for ERC20 token
        Should correctly update votes after BURN for ERC20 token
    Access control
        Should revert when NON-ADMIN grants role
        Should revert when NON-ADMIN calls #revokeRole
        Should revert when NON-MINTER mints tokens
        Should revert when NON-BURNER burns tokens
        The DEFAULT_ADMIN should be allowed to perform the #grantRole
        Should allow DEFAULT_ADMIN to revoke his role
        Should allow DEFAULT_ADMIN to revoke the admin role of another admin
    ZeroVotingERC721
      Should correctly set name and symbol for ERC721 token
      tokens should NOT be transferrable
      Should allow minter to #safeMint a token with proper URI
      Should emit Transfer event on #safeMint
      Should allow admin to #setBaseURI
      Should emit BaseURIUpdated event when baseURI is updated
     Voting functions
        Should delegate votes for ERC721 token
        Should correctly update votes after BURN for ERC721 token
  Access control
     Should revert when NON-ADMIN grants role
     Should revert when NON-ADMIN calls #revokeRole
     Should revert when NON-MINTER mints tokens
     Should revert when NON-BURNER burns tokens
     Should revert, if NON-MINTER tries to #safeMint
     Should NOT allow NON-ADMIN to #setBaseURI
     Should not allow non-admin to set token URI
     The DEFAULT_ADMIN should be allowed to perform the #grantRole
     Should allow revocation of ADMIN_ROLE role to YOURSELF
     The DEFAULT_ADMIN should be allowed to perform the #revokeRole
     Should allow admin to set token URI

223 passing (19s)
1 pending

Done in 20.64s.
```

8.2.1 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

8.2.2 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

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