Security Review Report
NM-0377 VerioIPA

NETHERMIND
SECURITY

(Feb 14, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for VerioIPA contracts. This security engagement focused on reviewing **VerioIPA**, a protocol that enables restaking vIP on IP Assets. The protocol allows users holding vIP tokens to stake them against IP assets, helping to attest and secure their validity on the network. The audit was conducted on the following commit 679d0f6.

The audited code comprises 2288 lines of code written in the Solidity language. The audit focused on the new IP Asset restaking functionality introduced by Verio.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** fifteen points of attention, where two are classified as `Critical`, one is classified as `High`, two are classified as `Medium`, two are classified as `Low` and eight are classified 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.

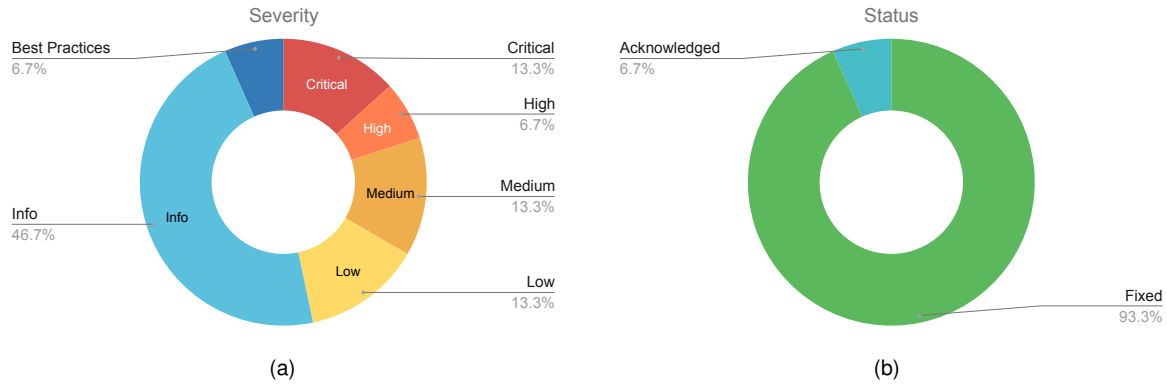

Fig. 1: Distribution of issues: Critical (2), **High** (1), **Medium** (2), **Low** (2), **Undetermined** (0), **Informational** (7), **Best Practices** (1). **Distribution of status: Fixed** (14), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Feb 5, 2025
Response from Client	Regular responses during audit engagement
Final Report	Feb 14, 2025
Repository	VerioIPA
Commit (Audit)	679d0f6f20b552c5740432bfdf20eb0fd1804a71
Commit (Final)	8d07a67f27a9eb9c20fb2954a50d66342110d517
Documentation Assessment	Medium
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/Lockup.sol	62	25	40.3%	19	106
2	src/ComponentSelector.sol	196	45	23.0%	32	273
3	src/IPAssetStaking.sol	340	90	26.5%	52	482
4	src/IncentivePool.sol	255	66	25.9%	37	358
5	src/CreatorRegistry.sol	64	26	40.6%	20	110
6	src/IPAssetStakePoolRegistry.sol	54	22	40.7%	16	92
7	src/OperatorRegistry.sol	137	39	28.5%	28	204
8	src/StakePool.sol	450	111	24.7%	72	633
9	src/abstract/Versionable.sol	16	9	56.2%	7	32
10	src/abstract/StakeToken.sol	30	12	40.0%	9	51
11	src/abstract/VerioComponent.sol	72	28	38.9%	23	123
12	src/abstract/AdminControlled.sol	17	12	70.6%	8	37
13	src/tokens/IP.sol	18	7	38.9%	4	29
14	src/tokens/VIP.sol	18	7	38.9%	4	29
15	src/interfaces/IIPAssetStaking.sol	98	127	129.6%	35	260
16	src/interfaces/ICreator.sol	13	13	100.0%	4	30
17	src/interfaces/IVerioComponent.sol	13	14	107.7%	7	34
18	src/interfaces/IStakePool.sol	129	150	116.3%	43	322
19	src/interfaces/IComponentSelector.sol	53	60	113.2%	22	135
20	src/interfaces/IOperator.sol	15	19	126.7%	4	38
21	src/interfaces/IAdminControlled.sol	10	13	130.0%	5	28
22	src/interfaces/IVersionable.sol	12	14	116.7%	6	32
23	src/interfaces/IIncentivePool.sol	50	66	132.0%	16	132
24	src/interfaces/IStakeToken.sol	24	30	125.0%	10	64
25	src/interfaces/IIPAssetStakePoolRegistry.sol	14	20	142.9%	8	42
26	src/interfaces/ILockup.sol	32	47	146.9%	17	96
27	src/interfaces/IOperatorRegistry.sol	32	58	181.2%	23	113
28	src/interfaces/IRewardPool.sol	39	29	74.4%	6	74
29	src/interfaces/ICreatorRegistry.sol	19	31	163.2%	13	63
30	src/interfaces/VerioIP/IVerioIPStakePool.sol	6	7	116.7%	3	16
	Total	**2288**	**1197**	**52.3%**	**553**	**4038**

3 Summary of Issues

	Finding	Severity	Update
1	A single malicious reward token can DoS the `StakePool` operations	Critical	Fixed
2	Repeated stake and unstake operations artificially increase the user's IP weight	Critical	Fixed
3	`NATIVE` token rewards cannot be claimed and will be stuck in the `IncentivePool`	High	Fixed
4	The `RewardPoolState` can be overridden, which will cause previous rewards to be lost	Medium	Fixed
5	The incentive pool creators can collude to DoS a `StakePool`	Medium	Fixed
6	Operator can steal user's rewards by setting the commission to 100 percent	Low	Acknowledged
7	Operators are unable to change their commission fees and reward addresses	Low	Fixed
8	Admin can update the lockup configuration affecting the currently locked funds	Info	Fixed
9	Authorized creators can register incentive pools on behalf of other users, which does not let the legitimate creator change the pool's config	Info	Fixed
10	If epoch duration is ever different than `1` , users will be losing rewards on each claim	Info	Fixed
11	The `IP` tokens sent on top of the `VerioIP` stake will be lost	Info	Fixed
12	The `RewardPool` can be registered with `0` reward tokens per epoch	Info	Fixed
13	When adding rewards, only one token type can be added at a time	Info	Fixed
14	`IPAssetStaking` contains functions open to re-entrancy attack vectors	Info	Fixed
15	Unused code	Best Practices	Fixed

4 System Overview

VerioIPA is a protocol built on top of **VerioIP** to enable restaking of **vIP** tokens on IP (intellectual property) assets. The system allows creators to register their IP within the protocol, and enables users to stake vIP (or other supported tokens) to validate, secure, and attest to IP ownership and usage. The protocol also features reward distribution mechanisms, incentive pools, time-based lockups, and operator delegation. Below is a high-level description of the major components in the **VerioIPA** architecture and the typical user flows.

The following sections delve into the system's components and their interactions.

4.1 IP Asset Staking

- Coordinates the core logic of staking tokens on IP assets.
- Users (or operators acting on their behalf) can stake tokens (vIP, IP, or other supported tokens) and optionally select a lockup duration (instant, short, long).
- Distributes rewards from incentive pools or reward pools when a user claims.
- Integrates with an IP's dedicated stake pool, as recorded in IIPAssetStakePoolRegistry.

4.2 IP Asset Stake Pool Registry

- Maps each IP asset to its respective stake pool contract.
- Each IP can have a unique stake pool that tracks staked tokens, lockup configurations, and user balances.
- Facilitates the creation or lookup of stake pools for newly registered IP assets.

4.3 Stake Pool

- Instantiated per IP asset, it manages user stakes, uncapped or time-locked positions, and reward distribution for that specific IP.
- Interacts with the ILockup contract to respect lockup durations and multipliers.
- Maintains a record of each user's stake amounts (by operator, token type, and lockup type), along with each user's last stake block.
- Can house multiple incentive pools from different creators.

4.4 Incentive Pool

- Allows creators or other sponsors to set up custom reward distributions to encourage staking on a particular IP asset.
- Supports multiple reward tokens, each with its own distribution logic (continuous, static, etc.).
- Users staking in the associated IStakePool can claim from these incentive pools based on their staked balance and stake duration.

4.5 Reward Pool

- Generic container for distributing rewards, typically in ERC20 or native tokens.
- Defines the reward token type and distribution model (e.g., continuous or static) to compute how much each staker earns over time.
- Various incentive pool(s) may use the same reward pool format to handle reward tokens.

4.6 Lockup

- Defines different lockup types (e.g., instant, short, long) with configurable staking multipliers and periods.
- Stake pool contracts reference the lockup details to enforce minimum stake durations or distribute rewards proportionally to the multiplier.
- Allows the protocol to introduce time-based staking mechanics without requiring each stake pool to implement them independently.

4.7 Operator / Operator Registry

- Provides an abstraction for "operators" who can stake, unstake, or claim rewards on behalf of end users.
- Each operator has a commission fee and a designated rewards address.
- The registry enforces certain constraints (e.g., max commission) and ensures that only valid operators act on user positions.

4.8 Creator / Creator Registry

- Tracks creator addresses who set up and manage incentive pools for their IP assets.
- llows for easy lookup of “incentive pool creators,” ensuring only authorized accounts can configure or fund a particular incentive pool.

4.9 Stake Tokens

- These tokens can be staked on IP assets. Two tokens will be supported in the initial stages of the protocol. **Native IP**, a native stake token representing the base layer currency and **vIP (ERC20 token)**, a synthetic or staked version of IP (obtained from the original VerioIP protocol).
- Each token implements a priceInIP function so the protocol can evaluate stake amounts in common units (IP).

4.10 VerioIP Stake Pool

- Provides bridging logic for conversions between IP and vIP in the broader Verio ecosystem.
- Used to calculate how many IP tokens correspond to a given amount of vIP when staking or unstaking.

4.11 Component Selector and VerioComponent

- Central mechanism for upgradability and versioning within VerioIPA.
- Tracks addresses for the IP Asset Registry, IP Asset Staking, lockup, operator registry, creator registry, and more.
- Allows an admin to upgrade individual modules without disrupting the entire system.
- Each component also implements IVersionable to record semantic version data and timestamps for auditability.

4.12 User Flows

IP Registration

- A creator or user calls the IIPAssetRegistry to register a new IP.
- If a registration fee is required, the user pays it to the treasury.
- The registry creates a dedicated IP Account (EIP-6551) and returns the new IP’s canonical address.

Stake Pool Setup

- Once an IP is registered, the system automatically or on-demand deploys a stake pool (via IIPAssetStakePoolRegistry).
- The stake pool references ILockup, IIncentivePool, and IRewardPool contracts for reward distribution and time-based locking.

Staking

- A user acquires vIP (or another allowed token) and calls IIPAssetStaking (or directly the IStakePool) to stake tokens on an IP asset.
- The user picks a lockup type (e.g., instant, short, or long) which affects multipliers and unbonding rules.
- The stake pool updates the user’s stake balance, records the block number, and updates total stake amounts for the IP asset.

Incentive Pools and Reward Distribution

- Creators or third parties can create IIncentivePool instances for their IP, specifying reward schedules or deposit tokens.
- Stakers periodically claim rewards from these pools, receiving distribution amounts proportional to their stake, lockup type, and staking duration.
- The system calculates how many IP or other tokens each user is entitled to, and the user can withdraw them at any time or after a required claim interval.

Operators

- A user or creator may designate an operator in the IOperatorRegistry.
- The operator (e.g., a platform or service) can stake, unstake, or claim on behalf of the user, often in exchange for a commission.
- The user’s rewards are split accordingly; a portion is paid to the operator’s reward address.

Unstaking and Withdrawal

- To unstake, a user (or operator) calls the unstake function on the relevant stake pool, specifying the amount and lockup type.
- Once unstaked, tokens may become immediately available (for instant lockups) or remain locked if a longer period was chosen.
- When fully unlocked, the user can withdraw tokens from the stake pool or claim them through the bridging logic if the token is vIP.

Upgrade and Versioning

- Each contract in the system implements IVersionable to track its semantic version.
- The IComponentSelector orchestrates upgrades, allowing the admin to replace a contract's address while preserving state.
- This architecture reduces downtime, enables fine-grained patches, and maintains an audit trail of changes.

In summary, VerioIPA extends the VerioIP staking paradigm by letting users restake vIP tokens (or stake IP directly) on registered IP assets. Through the combination of lockups, incentive pools, operator delegation, and reward distribution, the protocol incentivizes active participation in securing and validating intellectual property on-chain. By leveraging a modular upgrade approach and a robust set of registries (for operators, creators, IP assets, and stake pools), VerioIPA aims to offer flexibility, transparency, and adaptability in evolving DeFi-based staking use cases around IP.

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] A single malicious reward token can DoS the `StakePool` operations

File(s): `src/IncentivePool.sol`

Description: On every `StakePool` operation such as `stake(...)`, `unstake(...)` and `claimRewards(...)` the internal `_claimRewards(...)` is called. This function iterates over every incentive pool and then over every reward pool to claim the rewards and send the reward token to the user. If one of the reward tokens happen to be malicious and for example revert on every transfer, then the entire call to `_claimRewards(...)` will revert. As a result staking/unstaking for a particular `IPAsset` will not possible. Since removing a reward pool or an incentive pool is not possible, the denial of service (DoS) is permanent.

Recommendation(s): Multiple approaches can be taken to fix the issue. One includes allowing the user to choose which reward tokens to claim. Another possibility includes skipping tokens that revert or adding an option for the admin to remove such tokens from the list.

Status: Fixed.

Update from the client: Fixed in commit: `dae5415`

We have added a try/catch block that will handle the DOS case, but unfortunately we can do nothing to prevent malicious reward offerers from upgrading their ERC20 contracts to revert on transfer. Users should be aware of the dangers of upgradeable contracts when selecting stake pools with potentially unsafe reward tokens.

6.2 [Critical] Repeated stake and unstake operations artificially increase the user's IP weight

File(s): `src/StakePool.sol`

Description: When a user stakes tokens on an `IPAsset`, the `stake(...)` function of the asset's associated `StakePool` is invoked. If the user's current stake for the specified `stakeToken` is `0`, it indicates that the user has not previously staked with this token. In such cases, the `stakeToken` address is added to the user's array of staked tokens. Additionally, if this is the first stake for a particular lockup period within the `StakePool` contract, the `stakeToken` address is also added to the contract-wide list of staked tokens. The `_stake(...)` function that handles the relevant state updates is shown in the code snippet below.

```
function _stake(
    // ...,
    address _user,
    IStakeToken _stakeToken,
    uint256 _amount,
    ILockup.Type _lockup
) internal {
    IStakePool.StakePoolState storage stakePoolState = _stakePoolState();
    if (
        stakePoolState.totalStakePerStakeTokenByLockup[_lockup][_stakeToken]
            == 0
    ) {
        // @audit Add the stake token address to the global list.
        stakePoolState.stakeTokens.push(_stakeToken);
    }
    // @audit Increase the total stake for that stake token.
    stakePoolState.totalStakePerStakeTokenByLockup[_lockup][_stakeToken] +=
        _amount;

    IStakePool.UserPeriodBasedStake storage userStake = stakePoolState
        .stakeByOperatorUser[_operator][_user].userPeriodBasedStake[_lockup];

    if (userStake.stakeByToken[_stakeToken] == 0) {
        // @audit If user's stake is 0, then the address
        // of the token is added to an array of user's tokens.
        userStake.stakeTokens.push(_stakeToken);
    }
    userStake.stakeByToken[_stakeToken] += _amount;
    // ...
}
```

The problem present in the `StakePool` contract is that there is no logic that would remove the stake tokens from the arrays. Both user and contract stake get decreased in the `_unstake(...)` function. However, when the stake reaches `0`, the stake tokens are not removed. If the user would stake again with the same token, since his total stake for that token is `0`, the stake token would be added to both arrays for the second time.

The two arrays of stake tokens are used in the reward computation logic. The `_getUserStakeWeightedInIPForPeriod(...)` and `_getTotalStakeWeightedInIPForPeriod(...)` functions compute the summed stake weights for individual users and the total for all users, respectively. The computation is done by iterating over the `stakeTokens` array and adding up the stakes for each token. Since the array might contain duplicated elements, the end result might be inflated. Both functions execute similar logic.

Only the `_getUserStakeWeightedInIPForPeriod(...)` is shown below for brevity.

```
function _getUserStakeWeightedInIPForPeriod(
    address _operator,
    address _user,
    ILockup.Type _lockup
) internal view returns (uint256) {
    // ...
    IStakePool.UserPeriodBasedStake storage userStake = stakePoolState
        .stakeByOperatorUser[_operator][_user].userPeriodBasedStake[_lockup];
    uint256 userStakeInIP = 0;

    // @audit-issue The array may contain duplicated elements.
    for (uint256 i = 0; i < userStake.stakeTokens.length; i++) {
        IStakeToken stakeToken = userStake.stakeTokens[i];
        uint256 multiplier =
            _selector().lockup().getLockupMultiplier(_lockup);

        // @audit-issue As a result they will be included
        // in the total stake multiple times.
        userStakeInIP += stakeToken.priceInIP(
            userStake.stakeByToken[stakeToken] * multiplier
        );
    }
    return userStakeInIP;
}
```

By repeatedly calling `stake(...)` and `unstake(...)`, the user can artificially increase his stake weight to earn more rewards. If the user is the only staker, the total stake weight will also be increased. If other users join later, their share in the total weight will be smaller, and as a result, they will receive fewer rewards.

Recommendation(s): One of the possible solutions to the problem includes removing the stake token from the `stakeTokens` array once the user's or contract's stake reaches the value of `0`. This way, tokens won't be accounted for twice during reward computations.

Status: Fixed.

Update from the client: Fixed in commit: `d36f955`.

6.3 [High] `NATIVE` token rewards cannot be claimed and will be stuck in the `IncentivePool`

File(s): src/IncentivePool.sol

Description: The current protocol implementation allows users to create a `RewardPool` with the token's `RewardTokenType` being either `ERC20` or `NATIVE`. Users can call the `addRewards(...)` function from the `IPAssetStaking` contract to add rewards to such a pool. They can supply either `ERC20` tokens or `NATIVE` tokens as rewards.

The problem present in the contract is that the `claimRewards(...)` function, responsible for reward claiming, handles only `ERC20` token transfers. More specifically, the `_sendReward(...)` function from the `IncentivePool` contract is supposed to transfer the reward tokens to the user, but it does not contain the logic to handle native token transfers.

```
// @audit-issue The _sendReward function distinguishes between ERC20 transfers
// to the user OR the operator, but does not transfer NATIVE token rewards.
function _sendReward(/*...*/) internal returns (uint256) {
    // ...
    if (_userStakeAmount > 0) {
        // ...
        if (rewardAmount > 0) {
            if (_operatorAddress == _user) {
                IERC20(rewardPool.rewardToken).safeTransfer(
                    _user, rewardAmount
                );
                // ...
            } else {
                // ...
                IERC20(rewardPool.rewardToken).safeTransfer(
                    _user, userRewardAmount
                );
                // ...
            }
            // ...
        }
    }
    // ...
}
```

Since there is no way for the admin or the users to withdraw the rewards from the `RewardPool` other than via `claimRewards(...)`, any native tokens supplied to the reward pool as rewards will be lost. These funds will be stuck in the smart contract.

Recommendation(s): Consider adding the logic to handle `NATIVE` token transfers to the `_sendReward(...)` function.

Status: Fixed.

Update from the client: Fixed in commit 62c8d18

6.4 [Medium] The `RewardPoolState` can be overridden, which will cause previous rewards to be lost

File(s): src/IPAssetStaking.sol

Description: As per the protocol's design an individual user should be able to register only 1 `IncentivePool`. For 1 `IncentivePool`, however, the creator can register multiple `RewardPools` with various tokens and reward configurations. Pool registration is performed via the `registerRewardPool(...)` function from the `IPAssetStaking` contract.

In its current state, this function can be called multiple times by the same user. It allows the `IncentivePool`'s creator to override an existing pool with a new one. The following pool configuration is set whenever a new reward pool is created.

```
IRewardPool.RewardPoolState memory rewardPool = IRewardPool
    .RewardPoolState(
    _rewardPoolConfig.rewardToken,
    _rewardPoolConfig.rewardTokenType,
    _rewardPoolConfig.distributionType,
    _rewardPoolConfig.rewardsPerEpoch,
    0, // @audit rewards per token
    0, // @audit total rewards
    0, // @audit total distributed rewards
    block.number // @audit last epoch block
);

// @audit-issue Second registration of the same reward token
// will override the previous configuration.
incentivePoolState.rewardPoolByRewardToken[_rewardPoolConfig
    .rewardToken] = rewardPool;
```

This means that the `IncentivePool` creator can unknowingly or maliciously override the old reward pool with a new one, impacting the rewards distribution mechanism. This can happen whenever the creator attempts to create a new reward pool but will specify the same `rewardToken` as an already existent pool.

Overriding the pool will reset the `lastEpochBlock` used during rewards calculation. If this parameter gets reset, user rewards will also reset since no time elapsed to accrue rewards. Also, this allows the `IncentivePool` creator to change the type of the reward token from `ERC20` to `NATIVE` and vice-versa, change the distribution mechanism from `CONTINUOUS` to `STATIC` and vice-versa, or change the number of `rewardsPerEpoch`.

Furthermore, all tokens added as rewards to the previous rewards pool will be lost because the `totalRewards` parameter has also been reset.

Recommendation(s): Consider adding checks to ensure that the creator can override the pool only if all the rewards from the current pool have been claimed.

Status: Fixed.

Update from the client: Fixed in commit: e55bfed

6.5 [Medium] The incentive pool creators can collude to DoS a `StakePool`

File(s): src/StakePool.sol

Description: `Creators` have the ability to register `IncentivePools` for a particular `IPAsset` and they can also register `RewardPools` for each `IncentivePool`.

During reward claiming (which also happens upon staking/unstaking) there is a nested loop iteration over all `IncentivePools` and over all `RewardPools`.

`Creators` can maliciously or unknowingly register too many pools which will lead to out-of-gas reverts whenever users stake/unstake from a specific `StakePool`. The likelihood of this attack vector is Low since creators must be whitelisted by the Owner, but the impact is High since it will lead to a permanent DoS as you can't remove pools after they are added.

Recommendation(s): Consider adding a claiming mechanism that skips the tokens if they can't be transferred.

Status: Fixed.

Update from the client: Fixed in commit: c89984a

We have added a maximum number of incentive pools that can be added to any stake pool. Now if incentive pool creators collude, they will not be able to DOS the stake pool. They could still attempt to collude to create incentive pools with bad configurations for users (ie. low reward distribution). This is mitigated by our administration process, as we can always remove a misbehaving incentive pool creator.

6.6 [Low] Operator can steal user's rewards by setting the commission to 100 percent

File(s): src/OperatorRegistry.sol

Description: Operators can specify a commission fee that users have to pay whenever they stake/unstake through the said Operator. The fee is expressed in basis points (up to 10.000). The issue is that an operator can specify a fee of `10.000 bps` which means that there is no upper bound on the operator's commission.

Right before claiming the rewards on behalf of the user, an operator can increase their commission to `10.000 bps` and steal all the rewards in the form of a commission from the user. The severity is low since the user staked with the operator in the first place. The trust assumption however can be easily minimized by introducing an upper bound for the commission, say `2000 bps`.

Recommendation(s): Consider adding a limit to the maximum amount that an operator can claim as commission from users.

Status: Acknowledged.

Update from the client: Will not fix.

Stake provided by an operator on the user's behalf is still maintained separately from the user's own stake. Operators can claim 100% of the rewards from their own stake, but cannot claim rewards that the designated user is otherwise owed.

It is permissionless for an operator to stake on a user's behalf, so any stake provided therein is an act of good-faith and should not be considered to *belong* to the original user.

6.7 [Low] Operators are unable to change their commission fees and reward addresses

File(s): src/OperatorRegistry.sol

Description: The `setOperatorCommissionFee(...)` function from an `OperatorRegistry` contract allows an operator to change their `commissionFee`. The issue is that the function calls the internal `_tryPayOperatorUpdateFee` function, which requires the `msg.value >= operatorUpdateFee`. Because the `setOperatorCommissionFee` function is not marked as `payable`, operators calling them will have their transactions revert since the `msg.value` will always be `0`. Since zero is lower than the required `operatorUpdateFee`, the check will revert. The same issue is present in the `setOperatorRewardsAddress(...)` function.

Recommendation(s): Consider adding the `payable` keyword to the `setOperatorCommissionFee` and `setOperatorRewardsAddress` functions.

Status: Fixed.

Update from the client: Fixed in commit: `7dd11ff`

6.8 [Info] Admin can update the lockup configuration affecting the currently locked funds

File(s): src/LockUp.sol

Description: Whenever users stake, they have to choose one `LockUp` period provided by the protocol. The issue is that this period can be changed by the admin after users have already staked. For example, the lockup period can be increased to a very long time, or the multipliers provided by the `LockUp` period can be changed. This behavior violates the terms to which the user agreed during staking. The lockup type can be also removed entirely which will cause user funds to be stuck;

Recommendation(s): Consider adding a delay period before updating LockUps so that users have time to react and decide if they still want to keep their tokens staked after the change takes place.

Status: Fixed.

Update from the client: Fixed in commit: `3d2eaa4`.

We have implemented a time-lock mechanism for updates to our Lockup contract. The effective time of an update is determined by the previous lockup time. Users are now notified via events that an update is pending, and can choose to unstake before that change goes into effect.

Update from Nethermind Security: It is still possible for an Admin to change the instant lockup to something much longer. The effective update time is based on the previous lockup time, but since in the case of instant lockup this time is `0`, the change can become effective instantly.

Update from the client: Fixed in commit: `8d07a67`.

6.9 [Info] Authorized creators can register incentive pools on behalf of other users, which does not let the legitimate creator change the pool's config

File(s): src/StakePool.sol

Description: The `registerIncentivePool(...)` from the `IPAssetStaking` contract is used to register a new incentive pool for a particular `IPAsset`. It can only be called by the `IPAsset` owner or an authorized incentive pool creator whitelisted in the `CreatorRegistry` contract. The caller provides the `IncentivePoolConfig`, which will be passed to the `registerIncentivePool(...)` from the `StakePool` contract. A new incentive pool will eventually be deployed and initialized with the provided configuration.

The creation of the pool is credited to the `creator` address provided in the `IncentivePoolConfig` struct and not to the original caller of the `registerIncentivePool(...)` from the `IPAssetStaking` contract. The problem with this approach is that these two addresses can be different and possibly even belong to a user who is not the `IPAsset` owner or registered as a pool creator.

The most significant caveat includes a possibility where someone calls `registerIncentivePool(...)` on behalf of another creator who wants to register the incentive pool on their own. The legitimate creator's registration will revert since they are already marked as owners of an incentive pool. As a result, they can't change the pool's configuration, which is immutable after deployment.

Recommendation(s): Consider validating that the `msg.sender` in the initial call to `registerIncentivePool(...)` from the `IPAssetStaking` contract is equal to the `creator` address provided in the pool config.

Status: Fixed.

Update from the client: Fixed in commit: 47c81fc

6.10 [Info] If epoch duration is ever different than 1, users will be losing rewards on each claim

Remarks about the finding 6.10

The following finding has been found during the fix review phase as a result of changing the reward calculation logic in commit 62c8d18.

File(s): src/IncentivePool.sol

Description: Whenever reward claiming logic is triggered, the `_claimRewards(...)` function from the `IncentivePool` contract is executed. It first calls `_distributeRewards(...)` to compute and update the global reward per token increment, and then it calls `_sendRewards(...)` to send the reward tokens to the user.

```
function _distributeRewards(
    uint256 _totalStakedInPool,
    address _rewardToken
) internal returns (uint256) {
    // ...
    uint256 blocksPassed = block.number - rewardPool.lastEpochBlock;
    if (
        blocksPassed > 0
            && rewardPool.totalDistributedRewards < rewardPool.totalRewards
    ) {
        // ...
        rewardPool.rewardPerToken += rewardPerTokenIncrement;
        // ...
        // @audit If the user claimed at block number 3, epoch duration is 2,
        // and previous lastEpochBlock was 0, then the lastEpochBlock will
        // be set to (3 - 0) / 2 * 2 = 2.
        rewardPool.lastEpochBlock += (
            blocksPassed / incentivePoolState.epochDuration
        ) * incentivePoolState.epochDuration;
    }
    return rewardPool.rewardPerToken;
}
```

The `_sendRewards(...)` function calls the `_calculateUserRewardAmount(...)` function to determine the reward amount to send to the user. The computation logic is similar to the one in `_distributeRewards(...)`.

```solidity
function _calculateUserRewardAmount(
    uint256 _userStakeInIP,
    uint256 _userRewardPerToken,
    uint256 _totalStakedInPool,
    IRewardPool.RewardPoolState storage rewardPool
) internal view returns (uint256) {
    uint256 rewards = 0;
    // ...
    // @audit _distributeRewards(...) updated lastEpochBlock to 2,
    // as a result blocksPassed is 1.
    uint256 blocksPassed = block.number - rewardPool.lastEpochBlock;
    // @audit Since blocksPassed is 1 and there are undistributed rewards,
    // the execution will enter the if statement.
    if (
        blocksPassed > 0
            && rewardPool.totalDistributedRewards < rewardPool.totalRewards
    ) {
        rewards = rewardPool.distributionType
            == IRewardPool.DistributionType.STATIC
            // @audit-issue The rewards amount will be zero since 1 divided by 2
            // will be rounded down to 0.
            ? (blocksPassed / incentivePoolState.epochDuration)
                * rewardPool.rewardsPerEpoch
            : rewardPool.totalRewards - rewardPool.totalDistributedRewards;

        if (rewards == 0) {
            // @audit-issue The early return is triggered. The reward amount
            // to be sent to user is zero.
            return 0;
        }
        // ...
    }
    // ...
}
```

In the usual scenario (when epoch duration is one block), the `_calculateUserRewardAmount(...)` wouldn't recompute the reward distribution since `lastEpochBlock` is the same as `block.number` and `blocksPassed` evaluates to zero. This, however, is not the case when the epoch duration is different than one. Whenever that happens, the reward amount will be rounded to zero and returned to the `_sendRewards(...)` function.

The rewards won't be sent to the user, but the user's `rewardPerTokenByRewardToken` will be updated with the new reward per token incremented computed in the `_distributeRewards(...)`. As a result, the user lost the ability to claim the rewards for that period.

The impact of this issue would be high since users wouldn't be getting the rewards. However, as confirmed with the Verio team, this scenario can't happen since the `epochDuration` shouldn't ever be set to a value different than one. As a result, the finding is marked with informational severity.

Recommendation(s): If epoch duration must ever be set to a different value than one, consider reverting the changes introduced in commit 62c8d18 and replace the call to `_calculateUserRewardAmount(...)` with an inline reward calculation.

Status: Fixed.

Update from the client: Fixed in commit 8d07a67.

6.11 [Info] The IP tokens sent on top of the VerioIP stake will be lost

File(s): src/IPAssetStaking.sol

Description: The stake(...) function in the IPAssetStaking allows users to specify an amount of ERC20 tokens to deposit and msg.value to send if they attempt to deposit NATIVE tokens. The issue in the internal _isValidStakeTokenAmount function is that if the user attempts to deposit a valid amount of ERC20 tokens and erroneously sends a non-zero msg.value, the function will return true. This results in a loss of funds for the user because the NATIVE tokens are not accounted for when depositing ERC20 tokens.

```
function _isValidStakeTokenAmount(
    address _stakeTokenAddress,
    uint256 _amount
) internal view returns (bool) {
    // ...
    if (stakeTokenType == IStakeToken.Type.ERC20) {
        // @audit-issue missing `&& msg.value == 0` check.
        return _amount >= minStakeAmount;

    } else if (stakeTokenType == IStakeToken.Type.NATIVE) {
        return msg.value >= minStakeAmount && _amount == 0;

    } else {
        return false;
    }
}
```

Recommendation(s): The _isValidStakeTokenAmount function should also check that msg.value == 0 when users attempt to stake ERC20 tokens.

Status: Fixed.

Update from the client: Fixed in commit: 9b619ad

6.12 [Info] The RewardPool can be registered with 0 reward tokens per epoch

File(s): src/IPAssetStaking.sol

Description: The registerRewardPool(...) function from the IPAssetStaking contract can be used by IPAsset owners or authorized pool creators to register a new RewardPool. The problem with this function is that it does not validate the provided RewardPoolConfig. As a result, it is possible that, mistakenly or intentionally, a pool with zero rewardsPerEpoch can be created. Such a pool will be iterated over during reward claims, but it will never distribute any rewards and will waste the user's gas.

Recommendation(s): Consider adding missing input validation to the registerRewardPool(...) function.

Status: Fixed.

Update from the client: Fixed in commit: ed496a6

6.13 [Info] When adding rewards, only one token type can be added at a time

File(s): src/IncentivePool.sol

Description: The addRewards(...) function from the IPAssetStaking contract allows users to add reward tokens to the RewardPool. The accepted token types include NATIVE and ERC20.

```
function _addRewards(
    IStakePool _stakePool,
    address _incentivePoolCreator,
    uint256 _amount,
    address _rewardToken
) internal {
    // @audit-issue msg.value can be non-zero even when sending ERC20 tokens.
    _stakePool.addRewards{value: msg.value}(
        msg.sender, _incentivePoolCreator, _amount, _rewardToken
    );
}
```

It is not enforced that the msg.value must be 0 when adding ERC20 reward token types. The unaware user might want to combine a NATIVE token transfer with an ERC20 one, which would cause one of the two to be lost.

The `_handleAddRewards(...)` function from the `IncentivePool` contract that handles the token transfer from the user to the pool can only account for one token type being added as a reward at a time. The `msg.value` sent on top of the `ERC20` tokens would be lost, and native tokens would be stuck in the `IncentivePool` contract.

```
function _handleAddRewards(
    address _user,
    IRewardPool.RewardPoolState storage rewardPool,
    uint256 _amount
) internal returns (uint256) {
    if (rewardPool.rewardTokenType == IRewardPool.RewardTokenType.ERC20) {
        IERC20(rewardPool.rewardToken).safeTransferFrom(
            _user, address(this), _amount
        );
        // @audit-issue If msg.value was non zero, NATIVE tokens will be stuck
        // in the contract and not accounted for in the rewards.
        return _amount;
    } else if (
        rewardPool.rewardTokenType == IRewardPool.RewardTokenType.NATIVE
    ) {
        return msg.value;
    } else {
        revert InvalidRewardTokenType(rewardPool.rewardTokenType);
    }
}
```

Recommendation(s): Consider adding additional input validation to ensure that `msg.value` is `0` when adding `ERC20` tokens as rewards.

Status: Fixed.

Update from the client: Fixed in commit: `9b619ad`

We added a check to verify that the supplied amount of tokens is relative to its reward token type. This means that a user can still only supply one reward token at a time, but they will no longer erroneously lose native token supplied when specifying an ERC20 type reward token.

6.14 [Info] `IPAssetStaking` contains functions open to re-entrancy attack vectors

File(s): `src/IPAssetStaking.sol`

Description: The main functions of the `IPAssetStaking` contract, such as `stake(...)` and `unstake(...)` and their "on-behalf" variants, are protected with re-entrancy guards. These functions contain callbacks to the `msg.sender`, which can be used to re-enter other functions from the `IPAssetStaking` contract that lack the `nonReentrant` modifier. Examples of unprotected functions include the logic to add and claim rewards and register incentive and rewards pools.

Although no active re-entrancy exploits have been found in the codebase at its current state, the following is a list of possible attack vectors that might become active due to fixing other issues presented in this audit report.

The logic in the `claimRewards(...)` function performs an externall call to the reward token via `safeTransfer(...)` function. Any of the reward tokens could re-enter the `claimRewards(...)` function again before claimed reward details are updated. This is a bigger problem when handling `NATIVE` tokens. If the missing logic to handle `NATIVE` token rewards is added to the protocol, the `claimRewards(...)` function could be repeatedly re-entered to drain the `RewardPool` of `NATIVE` tokens. User's `rewardDetails` are updated after an external call that could be used to re-enter and claim again.

Other possible attack vector includes a call to `unstake(...)`. The `amount` of tokens to unstake is provided by the user and is not sanitized. Users can specify an arbitrarily large number higher than their original deposit. Before the actual unstake happens, the specified `amount` of funds will be withdrawn from the `StakePool` and sent to the user via the `withdraw(...)` function. Malicious users could use these funds to manipulate the state (e.g., add the funds as rewards). In its current form, the call to `unstake(...)` with an amount larger than the initial deposit would revert at the very end of the call chain due to arithmetic underflow during the user's stake token accounting updates.

Recommendation(s): Consider adding `nonReentrant` modifiers to the remaining functions in the `IPAssetStaking` contract to decrease the possible attack surface via re-entrancies.

Status: Fixed.

Update from the client: Fixed in commit: `22c914d`

6.15 [Best Practices] Unused code

File(s): `IPAssetStaking.sol`

Description: The function `_ipAssetRegisteredWithIPAssetRegistry(...)` from the `IPAssetStaking` contract is unused.

Recommendation(s): Consider removing the unused code from the contract to keep the codebase clean and readable.

Status: Fixed.

Update from the client: Fixed in commit: `52efe90`

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 on VerioIPA Documentation

The VerioIPA team actively provided documentation throughout the audit, sharing relevant information during the kick-off meeting and regular sync calls.
The team was highly responsive, addressing all questions during meetings and via messages, which greatly enhanced the auditing team's understanding of the project's technical aspects. However, maintaining concise written documentation of core flows would be beneficial for future security reviews.

8 Test Suite Evaluation

The Verio's test suite covers major flows of the protocol. The in-scope contracts could benefit from higher test coverage to make them more robust and less error-prone during future updates. To assess the test coverage for these files, the Nethermind Security team applied a technique called mutation testing to uncover the untested paths. This technique introduces slight modifications to the code called "mutations" or "mutants." An example of a mutation is, for example, changing the operator in an expression `(a + b)` to `(a - b)`, or removing a `require(a > b)` statement entirely from the code. With these changes, the code no longer follows the expected business logic of the application, and the test suite should reflect that by failing.

Evaluation of the test suite with mutation testing consists of two phases:

a. Generating the modified version of each contract, called "mutants."

b. Inserting the mutant into the original codebase and running the test suite.

Only one modification can be tested at a time. If the contract has ten mutations, the test suite must run ten times (once for every mutation). If any of the tests fail, it means that the test suite caught the change in the code. Whenever that happens, the particular mutant is considered "slain" or "killed" and is removed from the mutant's set. If that does not occur, a new test case can be added to cover the code branch to "kill" the mutant.

The following table outlines the results of the analysis performed on Verio's smart contracts. The first column lists the contracts that were tested using mutation testing. The second column indicates the number of mutants generated and how many were "slain" (i.e., caught by the test suite). The third column provides the percentage of mutants slain, reflecting the effectiveness of the test suite in covering the particular contract. The higher the score, the better the test suite is at finding bugs.

Contract	Mutants (slain / total generated)	Score
`OperatorRegistry.sol`	9 / 71	12.68%
`CreatorRegistry.sol`	4 / 24	16.67%
`StakePool.sol`	140 / 199	70.35%
`AdminControlled.sol`	1 / 3	33.33%
`IPAssetStakePoolRegistry.sol`	5 / 11	45.45%
`IPAssetStaking.sol`	120 / 242	49.59%
`IncentivePool.sol`	162 / 218	74.31%
`Lockup.sol`	12 / 65	18.46%
`StakeToken.sol`	1 / 4	25.00%
`VIP.sol`	0 / 7	0.00%
`ComponentSelector.sol`	8 / 68	11.76%
`VerioComponent.sol`	3 / 21	14.29%
`IP.sol`	0 / 7	0.00%
Total	465 / 940	49.47%

Core contracts such as `IPAssetStaking`, `StakePool`, and `IncentivePool` should be prioritized when writing new test cases. They play a crucial role in the system but are undertested.

8.0.1 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

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.