Security Review Report
NM-0377 VerioIP

NETHERMIND
SECURITY

(Feb 10, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for VerioIP contracts. The security engagement focused on reviewing **VerioIP**, a liquid staking protocol that enables users to stake IP and receive vIP (Verio IP tokens) in return. The protocol provides a secure and efficient way to participate in network validation while maintaining liquidity through tokenization. The audit was conducted on the commit b8ccd6e.

The audited code comprises 1881 lines of code written in the Solidity language. The audit focused on the implementation of the Verio liquid staking system.

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** nineteen points of attention, where one is classified as `Medium`, three are classified as `Low` and fifteen 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 test suite evaluation. Section 9 concludes the document.

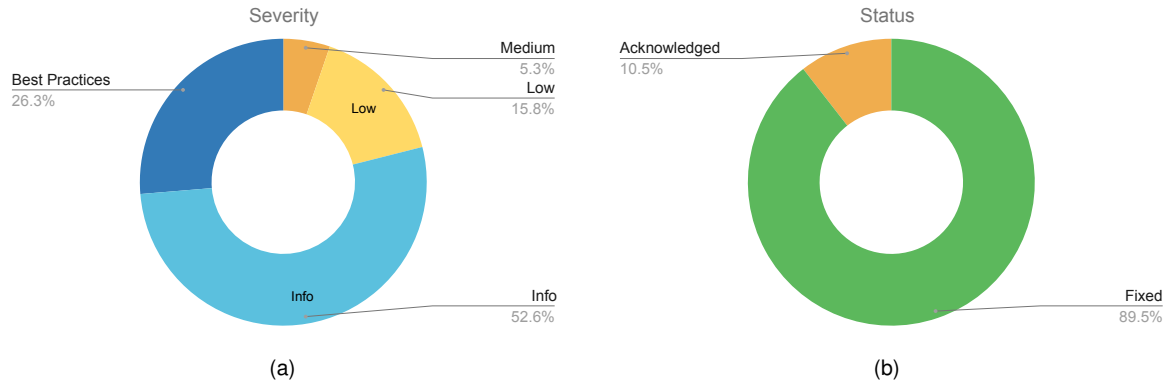

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (1), **Low** (3), **Undetermined** (0), **Informational** (10), **Best Practices** (5). **Distribution of status: Fixed** (17), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	January 27, 2025
Final Report	February 10, 2025
Repository	VerioIP
Commit (Audit)	b8ccd6e66cc36f1a219e247a0de77b6ffaaa3fc2
Commit (Final)	17343db17e68d9dd5380fd7d3259a87e7f602803
Documentation	NatSpec comments
Documentation Assessment	Medium
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/Delegator.sol	660	170	25.8%	106	936
2	src/VerioIP.sol	42	25	59.5%	18	85
3	src/WithdrawalPool.sol	116	32	27.6%	31	179
4	src/ComponentSelector.sol	198	39	19.7%	33	270
5	src/RewardPool.sol	65	26	40.0%	20	111
6	src/StakePool.sol	278	80	28.8%	70	428
7	src/InsurancePool.sol	68	31	45.6%	23	122
8	src/abstract/Versionable.sol	15	4	26.7%	7	26
9	src/abstract/AdminControlled.sol	17	5	29.4%	8	30
10	src/abstract/VerioComponent.sol	75	10	13.3%	24	109
11	src/interfaces/IAdminControlled.sol	9	2	22.2%	3	14
12	src/interfaces/IStakePool.sol	66	92	139.4%	38	196
13	src/interfaces/IVerioComponent.sol	11	2	18.2%	4	17
14	src/interfaces/IDelegator.sol	114	145	127.2%	48	307
15	src/interfaces/IWithdrawalPool.sol	26	37	142.3%	15	78
16	src/interfaces/IVersionable.sol	11	10	90.9%	5	26
17	src/interfaces/IVerioIP.sol	9	18	200.0%	6	33
18	src/interfaces/IRewardPool.sol	20	28	140.0%	12	60
19	src/interfaces/IComponentSelector.sol	57	71	124.6%	22	150
20	src/interfaces/IInsurancePool.sol	24	36	150.0%	15	75
	Total	**1881**	**863**	**45.9%**	**508**	**3252**

3 Summary of Issues

	Finding	Severity	Update
1	In certain cases, removing a validator will stop `Delegator` stake and unstake operations from finalizing until a new validator is added	Medium	Fixed
2	Potential for stuck funds when removing a `bondedMaturity`	Low	Fixed
3	The IP token withdrawals are rounded in the wrong direction	Low	Fixed
4	The `claimRewards(...)` function subtracts the fee twice	Low	Fixed
5	Adding, removing, and setting new distribution configs for bonded maturities will break the protocol's invariants	Info	Fixed
6	Calls to `initialize(...)` functions can be front-run	Info	Acknowledged
7	Execution cost of `stake(...)` is not constant for all users	Info	Fixed
8	If `unstakeInterval` is set too low, the limit of 14 concurrend unbounding requests can be exceeded	Info	Fixed
9	The `Minted` event is not emitted when minting `VerioIP` tokens to `treasury`	Info	Fixed
10	The `_handleAdminStake(...)` function does not emit the `DelegationStaked` event	Info	Fixed
11	The `_handleStake(...)` treats potential user refunds as funds to pay fees	Info	Fixed
12	The `redelegate(...)` will revert when called with more `attempts` than necessary	Info	Fixed
13	The bonded maturity unstake threshold check when processing the unstake queue is inaccurate	Info	Fixed
14	VerioIP `mint` and `burn` functions don't use the `whenNotPaused` modifier	Info	Acknowledged
15	Missing input validation	Best Practices	Fixed
16	The `withdraw(...)` function does not follow the Checks-Effects-Interactions pattern	Best Practices	Fixed
17	Lack of enforcement of `msg.value` inside `fund(...)` functions	Best Practices	Fixed
18	Lack of zero address checks in `ComponentSelector` setter functions	Best Practices	Fixed
19	No checks for `mintFeeBps` and `burnFeeBps` in the `StakePool` setter functions	Best Practices	Fixed

4 System Overview

The audited smart contracts implement a modular, upgradable staking and rewards solution for the **VerioIP** protocol. The system enables users to stake **IP tokens**, receive VerioIP **(vIP)** tokens, delegate stake to validators, claim or redistribute rewards, and withdraw staked tokens after an unbonding period. It also includes insurance coverage to protect certain components, subsidize the fees, and maintain system solvency. The core components in this architecture are managed through the **Component Selector** contract, ensuring each contract can be replaced or updated independently.

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

4.1 Component Selector

The Component Selector serves as the central registry and upgrade controller. It holds references to all the major contracts in the system—such as the Stake Pool, Delegator, Withdrawal Pool, Reward Pool, Insurance Pool, and VerioIP token contract and can update them individually. This design allows new implementations or versions of these components to be introduced without redeploying the entire system.

4.2 VerioIP Token

The VerioIP token represents the base IP asset. Authorized contracts (Stake Pool) can mint and burn it to reflect staking and unstaking operations. One of the core invariants of the protocol is that a single VerioIP token can always be exchanged back for at least one IP token. As rewards are added into the system, VerioIP tokens will appreciatete in price, and users will be able to exchange them for more IP tokens.

4.3 Stake Pool

The Stake Pool contract is the primary entry point for users who want to stake IP tokens. Users deposit IP and receive VerioIP (vIP) tokens in return. vIP tokens track their share of the staked balance. The contract supports flexible and/or locked staking (depending on configuration) and calculates mint/burn fees if enabled. Upon staking, the Stake Pool interacts with the Delegator contract to manage underlying validator delegations. When users unstake vIP, the Stake Pool calculates how many IP tokens they are entitled to get in return and initiates the unbounding flow.

4.4 Delegator

The Delegator is responsible for distributing staked assets across one or more validators and managing per-validator accounting, including different maturity periods (e.g., flexible vs. locked stakes). It maintains lists of validators, tracks total staked amounts per validator, and enforces bonding intervals or re-delegation logic. The Delegator can also process redeeming or redelegating stakes among validators in batches while respecting the system's configuration constraints.

4.5 Withdrawal Pool

When users request to unstake IP tokens, the funds go into the Withdrawal Pool. The Withdrawal Pool enforces unbounding periods: users' withdrawals become claimable only after a certain time passes. Once withdrawals are fully unbounded, they can be claimed. This design ensures a predictable lockup window and avoids immediate liquidity that could compromise the staking security model.

4.6 Reward Pool

The protocol rewards stakers for securing the network or contributing to validation. The Reward Pool accumulates rewards (in IP) that can be claimed by staking participants. The pool is funded through external or internal mechanisms (e.g., block rewards, protocol fees, or other reward sources). Upon specific calls, rewards are distributed proportionally to stakers, with an optional fee that is redirected to protocol treasury or other designated recipients.

4.7 Insurance Pool

The Insurance Pool provides coverage for specific failures or shortfalls in the system. It can also be used to subsidize the fees. Components such as the Stake Pool or Delegator may be designated "claimants." If a shortfall occurs—e.g., slashing or other unforeseen events, the affected component can claim funds from the Insurance Pool. The Insurance Pool maintains a target balance and can be funded by admins or through protocol fees. When coverage is provided to a claimant, the Insurance Pool's balance decreases accordingly.

4.8 Admin controls and Versioning

Several of the contracts mentioned above include an admin role for governance or emergency actions, such as updating fees, setting the maximum stake amount, or upgrading contract addresses. Each component also supports semantic versioning, allowing the protocol to track upgrades over time. By combining `IComponentSelector` and `IVersionable` interfaces, the system can cleanly roll out new contract versions in case of bug fixes or feature enhancements.

4.9 User Flows

Staking

- A user sends IP tokens to the Stake Pool via the `stake(...)` function.
- In exchange, the user receives `vIP` tokens representing their share of the `StakePool`.
- The `StakePool` routes IP to the `Delegator`, which distributes stake to configured validators according to maturity preferences and bonding rules.

Earning Rewards

- The Delegator or Stake Pool triggers reward collection from network emissions or other reward sources.
- The Reward Pool contract holds these rewards until they are claimed or automatically restaked.
- Users may benefit from increased balances or choose to redeem those rewards (depending on protocol settings).

Unbonding and Withdrawal

- When a user decides to unstake, they burn their vIP tokens in the Stake Pool.
- The protocol calculates how many IP tokens they are entitled to get in return and creates a withdrawal request in the Withdrawal Pool.
- After the unbonding period passes, the user can call withdraw, releasing IP tokens from the Withdrawal Pool to their address.

Insurance Coverage

- If a validator or other component suffers a slashing event or shortfall, it can request a payout from the Insurance Pool (if configured as a claimant).
- The Insurance Pool covers part or all of the shortfall, ensuring the protocol is solvent.
- Admins can replenish or adjust the Insurance Pool's target to maintain coverage adequacy.

Upgradability

- The admin can update individual components by calling setter functions on the `ComponentSelector` contract.
- Because each contract implements `Versionable`, the system's version history is recorded, and external verifiers can track changes to specific modules. This approach offers flexibility and reduces the scope of potential upgrade failures.

In summary, the **VerioIP** protocol is structured around a set of composable, upgradable contracts that collectively manage user stakes, rewards, and insurance. This architecture ensures that each core function (staking, delegation, rewards, insurance, and withdrawals) is cleanly separated, making the system modular and maintainable. Users benefit from a clear staking lifecycle (stake → earn → unbond → withdraw) and from protective features like the Insurance Pool. Meanwhile, the protocol can evolve over time through upgrades of the contracts in the **ComponentSelector**, minimizing disruption and risk to users.

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 [Medium] In certain cases, removing a validator will stop `Delegator` stake and unstake operations from finalizing until a new validator is added

File(s): `Delegator.sol`

Description: Whenever enough funds are gathered in the `StakePool` contract, the next stake operation will trigger the `handleStake(...)` function on the `Delegator` contract. Based on the current distribution, this function will deposit the tokens into an appropriate delegation maturity. The validator that will be chosen for this delegation is selected in the `_nextStakeValidator(...)` function. Once the validator is picked, the ID is updated for the validator that will be picked on the next stake operation.

```
function _nextStakeValidator() internal returns (bytes memory) {
    // ...
    // @audit Pick validator for the current delegation, based on the previously
    // set stakeIdx.
    bytes memory validatorPubkey =
        validatorRegistry.validators[validatorRegistry.stakeIdx];

    // @audit Update stakeIdx to prepare for the next time this function is
    // called. IDs iterate over the validators list in a round-robin fashion.
    validatorRegistry.stakeIdx = (validatorRegistry.stakeIdx + 1)
        % validatorRegistry.validators.length;

    return validatorPubkey;
}
```

The problem in the `_nextStakeValidator(...)` function is that, in certain cases, it might revert due to out-of-bound array access to the `validators` array.

Consider the following scenario: The `validatorRegistry.validators` array contains the following validators.

```
Validator: [A, B, C, D, E]
Array IDX:  0, 1, 2, 3, 4
```

During the last call to `_nextStakeValidator(...)`, the `validatorRegistry.stakeIdx` has been set to `4`. This means that validator `E` will be chosen for the next delegation. An issue arises if one of the validators is removed from the list before the next delegation happens. For this example, any validator can be removed, so let us assume that `B` is removed. This is the updated `validatorRegistry.validators` array.

```
Validator: [A, E, C, D]
Array IDX:  0, 1, 2, 3
```

As a result of calling `removeValidator(...)`, the validator `E` has been moved in the place of the removed `B`. The problem is that the `stakeIdx` still points to index `4`, which is no longer in the array. Once the `_nextStakeValidator(...)` function is called again to pick the validator, it will access an out-of-bound array index, and the execution will revert. The same problem is present in the `_nextUnstakeValidator(...)` function. Due to this, the `handleStake(...)` and `handleUnstake(...)` operations won't work until a new validator is added to the list.

Recommendation(s): Consider revisiting the `stakeIdx` logic so that removing the validator does not result in out-of-bound access.

Status: Fixed

Update from the client: Fixed in commit: `aedd7cb`. Now if removing a validator would result in an out-of-bounds index, the index is reset to 0. As a side effect of the removal process, the order of validators in the selection sequence may still be altered. A new check has also been added to ensure that there will always be at least 1 validator to stake with.

6.2 [Low] Potential for stuck funds when removing a `bondedMaturity`

File(s): `Delegator.sol`

Description: The `removeBondedMaturity` will remove a maturity from the `delegatorState.bondedMaturities` array without checking if that particular maturity still has funds deposited into it.

Users and validators that deposited their funds inside that maturity will have their funds stuck and they are unable to withdraw.

Inside `_handleBondedUnstakes(...)` function from the `Delegator` contract, we loop through the active bonded maturities array `(uint256 i = 0; i < delegatorState.bondedMaturities.length; i++)` before attempting to withdraw. If a certain bonded maturity is removed, then the protocol will not process withdrawals from it anymore.

Although the admin could re-add the same `bondedMaturity` with the same configuration in order to reenable withdrawals, this can lead to reputational damage if validators & users are temporarily unable to withdraw from the protocol.

Recommendation(s): Consider adding a check inside the `removeBondedMaturity(...)` to ensure that the maturity distribution is empty before removing it.

Status: Fixed

Update from client: Fixed in commit: `ae68927`. See *[Info] Adding, removing, and setting new distribution configs for bonded maturities will break the protocol's invariants* issue for details.

6.3 [Low] The IP token withdrawals are rounded in the wrong direction

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

Description: Whenever the users wish to exchange their `vIP` tokens back to IP tokens, they can call the `unstake(...)` function from the `StakePool` contract. Based on the provided `vIP` token amount, the `calculateIPWithdrawal(...)` function will compute the amount of IP tokens the user can withdraw from the `WithdrawalPool` after the unbonding period. The problem with the `calculateIPWithdrawal(...)` function is that it rounds up in favor of the user.

```
function calculateIPWithdrawal(uint256 _vIPAmount)
    public
    view
    override
    returns (uint256)
{
    uint256 totalStakedIP = _getTotalStake();
    uint256 totalSupply = _getTotalSupply();

    // ...

    // @audit The total IP after unstake is rounded down.
    uint256 totalIPAfterUnstake = (totalStakedIP + address(this).balance)
        * (totalSupply - _vIPAmount) / totalSupply;

    // @audit-issue As a result, the returned IP amount to withdraw
    // is slightly higher than it should be.
    return totalStakedIP + address(this).balance - totalIPAfterUnstake;
}
```

The rounding error can be considered a dust amount. In the most extreme scenario, if many deposits are assumed, after all users withdraw (with no inflow of rewards), the last user to withdraw will be able to get back slightly less than his initial deposit. The invariant in which `1 vIP` token is worth at least `1 IP` token will be broken.

Recommendation(s): To prevent any form of insolvency issues, it is considered a best practice to always round in favor of the protocol. Consider rounding up the `totalIPAfterUnstake` calculation so that the end result is rounded down.

Status: Fixed

Update from the client: Fixed in commit: `92a7b44`. A new formula to calculate the `totalIPAfterUnstake` has been applied. In this version, the amount is always rounded up.

6.4 [Low] The `claimRewards(...)` function subtracts the fee twice

File(s): `RewardPool.sol`

Description: The `claimRewards()` function is supposed to transfer a `fee` to the treasury and check if the `insurancePool` has debt. If it has, the funds in the `RewardPool` are meant to cover as much as possible from the debt. If the balance of the `RewardPool` exceeds the debt, then the difference will be transferred as rewards to the `StakePool` contract.

The issue is that the `fee` is subtracted from the `address(this).balance` twice, so the contract artificially reduces its perceived balance. In some cases, this may prevent `RewardPool` from covering all the debt of the `InsurancePool` (although the contract does have all the necessary funds) and transfer excess rewards to the `StakePool`.

```
function claimRewards() external override nonReentrant onlyStakePool {
    // ...
    uint256 fee = address(this).balance * rewardPoolState.feeBps / 10000;
    // @audit Fee is first sent to the treasury.
    (bool success,) = payable(treasury).call{value: fee}("");
    require(success, "RewardPool: failed to transfer fee to treasury");

    uint256 insurancePoolDebt = insurancePool.getDebt();
    if (insurancePoolDebt > 0) {
        // @audit-issue Since fee was already sent; address.balance is
        // already lower. The fee is subtracted for the second time.
        uint256 balanceAfterFee = address(this).balance - fee;
        uint256 fundAmount = insurancePoolDebt > balanceAfterFee
            ? balanceAfterFee
            : insurancePoolDebt;

        insurancePool.fund{value: fundAmount}();
    }
    // ...
}
```

Recommendation(s): To have a correct representation of the actual balance of the contract `balanceAfterFee` should be equal to `address(this).balance` without subtracting the `fee` again.

Status: Fixed

Update from client: Fixed in commit: `544080a`.

6.5 [Info] Adding, removing, and setting new distribution configs for bonded maturities will break the protocol's invariants

File(s): Delegator.sol

Description: When deploying the protocol, the `_trySetDistribution(...)` function from the `Delegator` contract validates that the maturity durations are stored in ascending order and the BPS sum of all maturities is `10000`.

```
function _trySetDistribution(MaturityConfig[] calldata _maturityConfigs)
    internal
{
    // ...
    uint256 lastMaturityDuration = 0;
    uint256 bpsSum = 0;

    for (uint256 i = 0; i < _maturityConfigs.length; i++) {
        if (
            _maturityConfigs[i].maturity
                == IIPTokenStaking.StakingPeriod.FLEXIBLE
        ) {
            // ...
            // @audit BPS for every maturity is summed.
            bpsSum += _maturityConfigs[i].bps;
        } else {
            // @audit Maturities are enforced to be in ascending order.
            require(
                _maturityConfigs[i].duration > lastMaturityDuration,
                "Delegator: Maturity durations must be in ascending order"
            );
            // ...
            lastMaturityDuration = _maturityConfigs[i].duration;
            // @audit BPS for every maturity is summed.
            bpsSum += _maturityConfigs[i].bps;
        }
    }
    // @audit Total distribution must equal 100%
    require(bpsSum == 10000, "Delegator: Maturity bps must sum to 10000");
    // ...
}
```

The problem is that the other functions that modify the maturity configuration, such as `addBondedMaturity` and `removeBondedMaturity` functions, do not consider the `bps` being added or removed. For example, after the protocol has been deployed, it has four maturities `[FLEXIBLE, SHORT, MEDIUM, LONG]`, and their `bps sum == 10000`. If the admin adds a new maturity, the `addBondedMaturity` doesn't redistribute the bps to accommodate the newly added maturity and enforces that their sum is still `10000 bps`. The sum of the maturities will exceed 10.000 bps. The same thing happens when maturity is removed via `removeBondedMaturity` function.

A similar check is missing inside the `setDistributionConfigByMaturity(...)` function, where the admin can change the configs for a particular distribution. After updating the configs, the sum of all distributions should be 10000 bps.

In addition, the `removeBondedMaturity(...)` function will reorder the elements of the `bondedMaturities` array when the element that gets deleted is not the last one in the array.

Building on the previous example, if the admin removes `SHORT` bonded maturities, the array will look like this `[FLEXIBLE, LONG, MEDIUM]`. This breaks one of the assumptions about the protocol, where maturities are expected to be in ascending order. If this happens, the `_nextStakeDelegation` function will attempt to fill the "longest" maturity first, starting from the last index of the `bondedMaturities` array. As can be seen, the `LONG` duration is no longer the last one. It is in the middle index, so the maturity that gets filled first will be `MEDIUM`.

Recommendation(s): Consider adding logic to redistribute the bps of bonded maturities whenever new ones are added, old ones are removed, or when their params are updated via `setDistributionConfigByMaturity` and enforce that the sum is always 10.000.

Status: Fixed.

Update from client: Fixed in commit: `ae68927`. The setters for adding and removing individual bonded maturities, and their configs, have been removed. Now the entirety of the distribution config must be updated at once. This means that the basis point sum, and ascending order of maturities, will be checked each time.

Update from Nethermind Security: The newly added loop in the `_trySetDistribution(...)` function ensures that old maturity with stake must be present in the new config so that it is not lost. The problem is that the `continue` keyword from the inner loop does not "continue" in the outer loop. Even if old maturity is found in the new config, the inner loop will finish, and the `revert` will be hit every time. The iteration should be continued in the outer loop, but it is stopped by a revert. A boolean flag `found` could be set once a match is found. The outer loop would then revert only when `found` is false.

Update from client: Fixed in commit: `cdd9cf5`.

6.6 [Info] Calls to `initialize(...)` functions can be front-run

File(s): `src/*`

Description: All major protocol smart contracts are proxy contracts initialized via the `initialize(...)` functions. These functions are unrestricted and transfer the contract ownership to the `msg.sender`. The attacker may attempt to front-run a call to `initialize(...)`, hoping it will go unnoticed before the protocol starts its operations. If that would be the case, a malicious actor would have access to the admin functionalities.

Recommendation(s): Consider adding extra validation to the deployment scripts to ensure that the ownership of the contracts was granted to an appropriate address during the project's deployment.

Status: Acknowledged

Update from the client: Verio components are all `AdminControlled` and as part of our standard deploy process we try to set the admin for each component to be the deployer initially. If the initializer is front-run, this call will fail because the `setAdmin` function is only callable by each component's owner.

6.7 [Info] Execution cost of `stake(...)` is not constant for all users

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

Description: The main entry point to the system for the users is the `stake(...)` function in the `StakePool` contract. Individual users' deposits wait in the `StakePool` contract until their sum reaches a threshold above the minimum stake amount accepted by the Story's `IPTokenStaking` contract. Once this threshold is reached, the next call to `stake(...)` will call the `handleStake(...)` function on the `Delegator` contract and send all the individual deposits in a batch for staking. The caveat of this design is that the users whose stake happens to pass the threshold will pay more gas for the `stake(...)` transaction execution since they have to cover the cost of calling the `handleStake(...)` function.

Recommendation(s): While this design is correct and efficient, consider informing the users about such a possibility or adding a flag parameter that makes a call to `handleStake(...)` optional. If the latter is chosen, an additional function would be required for a protocol keeper to move the funds from the `StakePool` to the `Delegator` at regular intervals if no user decides to call `handleStake(...)`.

Status: Fixed

Update from the client: Fixed in commit: `12ccd18`. Users can now elect whether or not to evaluate the `stakePool` when calling stake. A new function `evaluateStakePool` has been added, and will be called by the Verio periodically to ensure continued protocol operation as expected.

6.8 [Info] If `unstakeInterval` is set too low, the limit of 14 concurrend unbounding requests can be exceeded

File(s): src/Delegator.sol

Description: The `unstakeInterval` variable in the `Delegator` contract dictates the frequency with which `unstake(...)` can be called on the Story's `IPTokenStaking` contract. As per Story's documentation, a single validator cannot have more than 14 concurrent unbounding requests at the same time. Any subsequent request will be ignored and will silently fail.

In a situation where the number of validator in the Verio protocol is low and the admin sets the `unstakeInterval` to a low value, the limit of 14 concurrent requests could be exceeded. This would result in a state mismatch between contract's state and the reality since delegations would be deleted from the unstake queue, but in reality they wouldn't be unstaked from the staking contract.

Recommendation(s): Consider setting a reasonable lower bound for the `unstakeInterval` in both the `setUnstakeInterval(...)` and `_trySetDelegatorState(...)` functions.

Status: Fixed

Update from the client: Fixed in commit: `0145136`.

6.9 [Info] The `Minted` event is not emitted when minting `VerioIP` tokens to `treasury`

File(s): src/StakePool.

Description: Whenever users stake funds with the `stake(...)` function, the `VerioIP` tokens are minted to their addresses. The `Minted(_staker, vIPToMint)` event is emitted with user's address and the amount of tokens minted. This amount however, does not include the `mintFee` amount of `VerioIP` tokens that were minted to the `treasury` address.

```
function _stake(address _staker, uint256 _stakeAmount) internal {
    // ...
    if (mintFee > 0) {
        // @audit-issue Treasury mint does not emit the Minted event.
        vIP.mint(treasury, mintFee);
    }
    vIP.mint(_staker, vIPToMint);
    emit Minted(_staker, vIPToMint);
}
```

Recommendation(s): Consider emitting the `Minted` event when minting `VerioIP` tokens to the treasury. The event could be added to the `VerioIP` token contract so that events are emitted every time, regardless of the token recipient.

Status: Fixed

Update from client: Fixed in commit: `a3efa4`. The `Minted` event has been moved to the `VerioIP` token contract to ensure it is properly handled for each `mint` call.

6.10 [Info] The `_handleAdminStake(...)` function does not emit the `DelegationStaked` event

File(s): src/Delegator.sol

Description: The `adminStake(...)` function from the `StakePool` contract can be used by an admin to inject liquidity at specific maturity for a specific validator. The stake will eventually be processed in the `_handleAdminStake(...)` function in the `Delegator` contract. However, unlike its regular `_handleStake(...)` variant, this function does not emit the `DelegationStaked` event when the delegation is processed.

Recommendation(s): Consider adding the missing event emission to the `_handleAdminStake(...)` function.

Status: Fixed

Update from the client: Fixed in commit: `7a48be9`.

6.11 [Info] The `_handleStake(...)` treats potential user refunds as funds to pay fees

File(s): `Delegator.sol`

Description: The `_handleStake()` function calls the `IPTokenStaking::stake` function. Inside this function the code calculates two values, a `stakeAmount` and a `remainder` by calling the `roundedStakeAmount` function.

```
function roundedStakeAmount(uint256 rawAmount) public pure returns (uint256 amount, uint256 remainder) {
    remainder = rawAmount % STAKE_ROUNDING;
    amount = rawAmount - remainder;
}
```

This function takes the amount and divides it by `STAKE_ROUNDING` which is equal to `1 gwei`. This means that if the actual amount is not perfectly divisible by 1 gwei, the difference will be treated as a `remainder`.

The `IPTokenStaking::_stake` function refunds the remainder to `msg.sender` which in the context of `IPTokenStaking` will be the `Delegator` contract.

```
function _refundRemainder(uint256 remainder) private {
    (bool success, ) = msg.sender.call{ value: remainder }("");
    require(success, "IPTokenStaking: Failed to refund remainder");
}
```

The `Delegator::receive()` function allows the contract to receive the refund, however this means that users who deposit amounts that are not perfectly divisible by 1 gwei will always incurr a small loss due to this rounding. The actual amount is negligible though.

Recommendation(s): In order to prevent this, a new check can be added in the `StakePool::stake` function that checks if `msg.value % 1 gwei == 0`.

Status: Fixed

Update from client: Fixed in commit: `a97f9ac`. `stake` now reverts if the supplied amount is not evenly divisible by 1 gwei.

6.12 [Info] The `redelegate(...)` will revert when called with more `attempts` than necessary

File(s): src/Delegator.sol

Description: When calling `redelegate(...)`, the admin has to specify the number of `attempts` to specify the number of redelegations to make. On every attempt, the function will dequeue the delegation from one validator and enqueue it for the other. At the end of each loop iteration, the Story's `IPTokenStaking` contract's `redelegate(...)` function will be called to move the delegation ID between validators.

```
function _redelegate(
    bytes calldata _fromValidator,
    bytes calldata _toValidator,
    uint256 _attempts
) internal returns (uint256 remainder) {
    // ...
    remainder = validatorRegistry.delegations[_fromValidator].total;
    for (uint256 i = 0; i < _attempts; i++) {
        // ...
        uint256 delegationId; // @audit these two vars are reset on each iteration
        uint256 amount;

        // @audit If there are no more delegations available, the following
        // if/else won't update delegationId and amount.
        if (fromValidatorDelegations.distribution.instant > 0) {
           // ...
           // process instant delegations
        } else {
           // ...
           // process bonded delegations
        }
        // ...
        // @audit-issue A call to redelegate will revert since
        // delegationId and amount are 0.
        ipTokenStaking.redelegate{value: fee}(
            _fromValidator, _toValidator, delegationId, amount
        );
        remainder = remainder > amount ? remainder - amount : 0;
    }
    return remainder;
}
```

An issue arises if the specified number of `attempts` exceeds the number of delegations available for a given validator. After all delegations are processed, the loop in the `_redelegate(...)` might attempt an extra iteration. Since there are no delegations to dequeue and enqueue, the `if/else` logic won't be executed, but the `IPTokenStaking` contract will still be called. The problem is that the `delegationId` and `amount` used as parameters to the `redelegate(...)` call will be uninitialized. Since the staking contract does not accept these values, the execution will revert, and the gas used up to this point will be wasted.

Recommendation(s): Consider terminating the loop early so that the execution does not revert when there are no more delegations to redelegate. Since the remaining amount to redelegate is stored in the `remainder` variable, one of the possible ways to fix the problem is to stop the loop execution once the `remainder` reaches `0`.

Status: Fixed

Update from the client: Fixed in commit: `7d1f7b3`. There is now an exit check that will return early if there are no more delegations to redelegate.

6.13 [Info] The bonded maturity unstake threshold check when processing the unstake queue is inaccurate

File(s): src/Delegator.sol

Description: Whenever the `handleUnstake(...)` function is called on the `Delegator` contract, the existing unstake queue will be processed. The `_handleUnstakeQueue(...)` first checks if the unstake interval has passed, and if so, it attempts to unstake the delegations from the Validators present in the unstake queue. A delegation can be unstaked if its amount exceeds the minimum stake threshold. Since instant unstakes are stored internally as one element, the threshold check in the `if` branch can check the total of instant unstakes. A similar check is performed in the `else if` branch for bonded unstakes. The relevant code snippet is shown below.

```
function _handleUnstakeQueue() internal {
    // ...
    for (uint256 i = 0; i < unstakeQueue.unstakeValidators.length; i++) {
        // ...
        // @audit Process instant unstake
        if (_stakeThresholdMet(instantUnstake.amount)) {
            // ...
            //     @audit-issue The condition below checks if total in queue
            //     is above the threshold. It should check if the delegation
            //     we are about to unstake is above the threshold instead.
        } else if (_stakeThresholdMet(bondedUnstakes.total)) {
            // @audit Proces bonded unstakes.
            Delegation memory bondedUnstake = bondedUnstakes.delegations[bondedUnstakes
                .ids[bondedUnstakes.ptr]];
            // ...
            ipTokenStaking.unstake{value: fee}(
                bondedUnstake.validator,
                bondedUnstake.id,
                // @audit A single delegation is unstaked,
                // not the total from the queue.
                bondedUnstake.amount,
                "" // Empty data parameter
            );
            // ...
        }
    }
    // ...
}
```

The `else if` condition is not accurate in the context of bonded maturities since they are unstaked one by one, unlike instant delegations, which are unstaked all in one go. To be accurate, the `else if` condition should check if the next delegation to unstake satisfies the threshold.

While in the current version of the code, this inaccurate check does not make a difference since all stakes/unstakes will have at least the threshold amount of stake, it might change once `rebase(...)` logic is implemented. If the validator's stake were to be decreased, then that particular delegation wouldn't satisfy the unstake condition on its own. Still, since the total in the queue could satisfy the check, the execution would continue.

Recommendation(s): Consider revisiting the condition to unstake bonded maturities in the context of possible future `rebases(...)`.

Status: Fixed

Update from the client: Fixed in commit: `b185546`. The threshold check here is not to see if the bonded delegations are of sufficient size, but rather to check that there are bonded delegations to unstake from the `unstakeQueue` before proceeding. The logic has been updated to reflect this intent.

6.14 [Info] VerioIP `mint` and `burn` functions don't use the `whenNotPaused` modifier

File(s): `VerioIP.sol`

Description: Because it inherits `PausableUpgradeable`, the `VerioIP` contract could be paused, but the `mint(...)` and `burn(...)` functions do not use the `whenNotPaused` modifier. The admin might want to pause token operations in case of emergency, but this won't be possible.

Recommendation(s): While minting can be controlled via `StakePool` which can be paused, users are still allowed to burn in the paused state. If it is desired, consider pausing burning as well.

Status: Acknowledged

Update from client: Will not fix. This is intentional behavior, as it still allows a user who owns VerioIP to burn it at will. Minting can be disabled by pausing the `StakePool` because it is the only valid caller of the mint function.

6.15 [Best Practices] Missing input validation

File(s): `src/*`

Description: Certain important variables in the contracts lack proper validation, which could lead to unexpected behaviors. Below is a non-exhaustive list of missing checks split per file.

`StakePool.sol`:

- `_trySetStakePoolState(...)`:;

- `minStake` can be `0` - `unbondingPeriod` can be `0`

- `setMinStake(...)`: `minStake` can be `0`;
- `setMintFeeBps(...)`: missing upper bound check;
- `setBurnFeeBps(...)`: missing upper bound check;

`ComponentSelector.sol`:

- all setter functions are missing `0` value checks;
- `_trySetComponentSelectorState(...)`: missing `0` value checks on the provided config;

`InsurancePool.sol`:

- `_trySetInsurancePoolState(...)`: missing `0` value checks on the provided config;
- `setClaimant(...)`: `claimant` can be `0`;

Recommendation(s): Consider adding extra input validation for the cases outlined above.

Status: Fixed

Update from the client: Fixed in commit: `f4e5c4b`. The only exception to this fix is the `unbondingPeriod`, which may be intentionally set to 0 to allow users to instantly withdraw.

6.16 [Best Practices] The `withdraw(...)` function does not follow the Checks-Effects-Interactions pattern

File(s): src/WithdrawalPool.sol

Description: Users can use the `withdraw(...)` function from the `WithdrawalPool` contract to withdraw their `IP` tokens. The function sends the tokens to the user via a low-level `call` and decreases the `totalPendingWithdrawals` afterward. Since this value could be modified before the external call, it is considered a best practice to respect the Checks-Effects-Interactions pattern and perform state updates before external calls.

Recommendation(s): To reduce the potential attack surface, consider moving the state update before the external call.

Status: Fixed

Update from the client: Fixed in commit: `74ab5aa`. Although this function is already `nonReentrant`, the state altering code has been moved before the external call.

6.17 [Best practices] Lack of enforcement of `msg.value` inside `fund(...)` functions

File(s): `StakePool.sol`, `WithdrawPool.sol`, `InsurancePool.sol`, `RewardPool.sol`

Description: The `fund()` functions in all the above contracts have no enforcement for `msg.value`. This allows any user to call these functions with `msg.value = 0` in order to spam and emit worthless events. There are legitimate situations in which the protocol will call these functions with potentially `msg.value = 0`. In order to differentiate between these cases, certain constrains can be added to the `fund(...)` functions. If the caller is one of the contracts of the system, allow `msg.value` to be zero, else revert. This will allow the system to work as expected, while preventing users from spamming the `fund()` function with 0 value.

Recommendation(s): Consider adding checks for `msg.value > 0` if `msg.sender` is not one of the protocol's smart contracts.

Status: Fixed

Update from client: Fixed in commit: `df260c3`.

6.18 [Best practices] Lack of zero address checks in `ComponentSelector` setter functions

File(s): `ComponentSelector.sol`

Description: None of the setter functions in the `ComponentSelector` contract check if the new `address` param is `address(0)` or not.

Recommendation(s): Consider adding this extra checks to the setter functions.

Status: Fixed

Update from client: Fixed in commit: `7c9d20b`.

6.19 [Best practices] No checks for `mintFeeBps` and `burnFeeBps` in the `StakePool` setter functions

File(s): `StakePool.sol`, `StakePool.sol`

Description: When the `StakePool` contract is deployed and initialized, the `_trySetStakePoolState` function has two checks for the `mintFeeBps` and `burnFeeBps` that enforce these fees to be at most 5%.

```
if (_stakePoolConfig.mintFeeEnabled) {
    require(_stakePoolConfig.mintFeeBps <= 500, "StakePool: max mint fee bps is 5%");
}

if (_stakePoolConfig.burnFeeEnabled) {
    require(_stakePoolConfig.burnFeeBps <= 500, "StakePool: max burn fee bps is 5%");
}
```

However, the setter functions for these fees, lack these checks. This allows the admin to potentially set higher fees than the expected maximum 5% threshold.

```
function setMintFeeBps(uint256 _mintFeeBps) external override onlyAdmin {
    //@audit no check for mintFeeBps <= 500
    StakePoolState storage stakePoolState = _stakePoolState();
    stakePoolState.mintFeeBps = _mintFeeBps;
    emit StakePoolStateUpdated(stakePoolState);
}
```

Recommendation(s): Consider adding these extra checks in the setter functions to maintain consistency.

Status: Fixed

Update from client: Fixed in commit: `c24f50c`. New functions have been added to vet the basis point configurations for both fees.

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 the VerioIP Documentation

The documentation for the VerioIP repository is primarily contained within NatSpec comments. Additional context was provided by the team during the kick-off meeting and regular sync calls.
The team consistently answered all questions during meetings and through messages, which gave the auditing team valuable insight and a deep understanding of the project's technical aspects. However, the project would benefit from concise written documentation covering all core flows to facilitate future security review engagements.

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
`RewardPool.sol`	19 / 49	38.78%
`ComponentSelector.sol`	10 / 71	14.08%
`Delegator.sol`	319 / 535	59.63%
`VerioIP.sol`	12 / 16	75%
`WithdrawalPool.sol`	65 / 94	69.15%
`InsurancePool.sol`	20 / 35	57.14%
`StakePool.sol`	197 / 337	58.46%
`AdminControlled.sol`	3 / 3	100%
`VerioComponent.sol`	7 / 23	30.43%
Total	657 / 1163	56.49%

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

8.1 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.