Security Review Report
NM-0628 Hyperbeat

NETHERMIND SECURITY

(Sep 02, 2025)

Contents

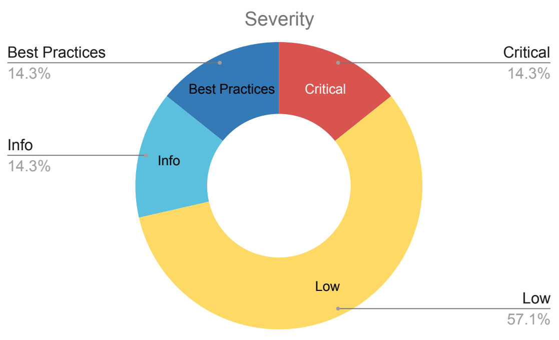
1 Executive Summary

This document presents the security review performed by Nethermind Security for Hyperbeat protocol smart contracts. Hyperbeat is a native protocol on Hyperliquid, purpose-built to scale the network and its ecosystem. It offers HYPE staking through its validator on Hyperliquid. `beHYPE` is a Hyperliquid-native liquid staking token integrated with the CoreWriter system contract. When users stake HYPE, the contract submits staking actions to HyperCore via CoreWriter and mints `beHYPE` token at the current exchange ratio.

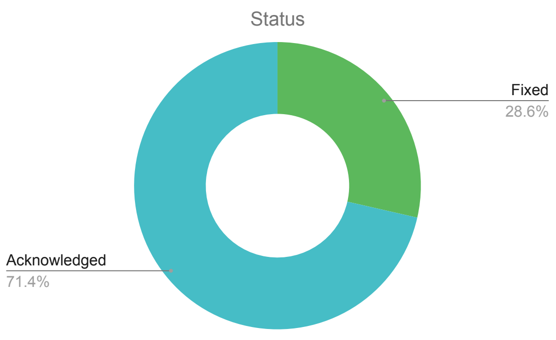
The audited code comprises 651 lines of code written in the Solidity language.

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



(a)



(b)

Fig. 1: Distribution of issues: Critical (1), **High** (0), **Medium** (0), **Low** (4), **Undetermined** (0), **Informational** (1), **Best Practices** (1).
Distribution of status: Fixed (2), **Acknowledged** (5), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	September 01, 2025
Final Report	September 02, 2025
Response from Client	Regular responses during audit engagement
Repository	beHYPE
Commit	2b9e23e1c6b98e45d93e97160c967a3665114d64
Final Commit	fb1c2d8cb3a7870752189db0f97791329a71a02e
Documentation	docs.hyperbeat, README, and Hyperliquid GitBook
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/StakingCore.sol	166	15	9.0%	61	242
2	src/BeHYPETimelock.sol	10	1	10.0%	4	15
3	src/BeHYPE.sol	49	4	8.2%	16	69
4	src/RoleRegistry.sol	81	31	38.3%	26	138
5	src/WithdrawManager.sol	204	17	8.3%	64	285
6	src/interfaces/IStakingCore.sol	37	142	383.8%	32	211
7	src/interfaces/IBeHYPE.sol	9	15	166.7%	6	30
8	src/interfaces/IRoleRegistry.sol	31	129	416.1%	29	189
9	src/interfaces/IWithdrawManager.sol	48	48	100.0%	23	119
10	src/interfaces/ICreate3Deployer.sol	16	1	6.2%	2	19
	Total	**651**	**403**	**61.9%**	**263**	**1317**

3 Summary of Issues

	Finding	Severity	Update
1	Malicious user can permanently block withdrawal finalizations	Critical	Fixed
2	Donations to `StakingCore` can lead to a temporary Denial of Service of `updateExchangeRatio(...)`	Low	Acknowledged
3	Instant withdrawal rate limit is stricter than intended	Low	Acknowledged
4	Just-in-Time liquidity strategy on `updateExchangeRatio()` can take yield from stakers	Low	Acknowledged
5	Precision loss during withdrawals can lead to insufficient funds and failed payouts	Low	Acknowledged
6	Queued withdrawal payout amount is fixed at the time of request and does not account for slashing	Info	Acknowledged
7	`finalizeWithdrawals()` violates the Checks-Effects-Interactions pattern	Best Practices	Fixed

4 System Overview

4.1 Hyperliquid Overview

Hyperliquid is a Layer-1 blockchain built from the ground up, crafted from first principles, and refined for real-world performance. It employs a custom consensus, HyperBFT, inspired by HotStuff and its successors.

Execution Architecture. The state execution is divided into two complementary components:

- **HyperCore** — a dedicated engine hosting on-chain perpetual futures and spot *order books*. Each order, cancel, trade, and liquidation is executed transparently on chain with one-block finality inherited from HyperBFT.
- **HyperEVM** — an EVM-compatible, general-purpose smart-contract runtime. Through HyperEVM, HyperCore's high-performance liquidity and financial primitives become permissionless building blocks for builders and users.

HyperCore Access: Querying State and Sending Actions

Read precompiles - The testnet EVM exposes read precompiles for reading from HyperCore. The addresses start at:

```
0x0000000000000000000000000000000000000800
```

These calls cover perp positions, spot balances, vault equity, staking delegations, oracle prices, and even the L1 block number.

CoreWriter contract - HyperEVM-to-HyperCore writes are sent through the CoreWriter system contract at:

```
0x3333333333333333333333333333333333333333
```

CoreWriter burns gas before emitting a log that is processed by HyperCore as an action. For example, a contract submits an action to HyperCore on behalf of its own address.

4.2 Hyperbeat Overview

Fig. 2 illustrates the communication diagram of the core contracts under audit. The main functions that the **User** interacts with the protocol are *stake* and *withdraw.*

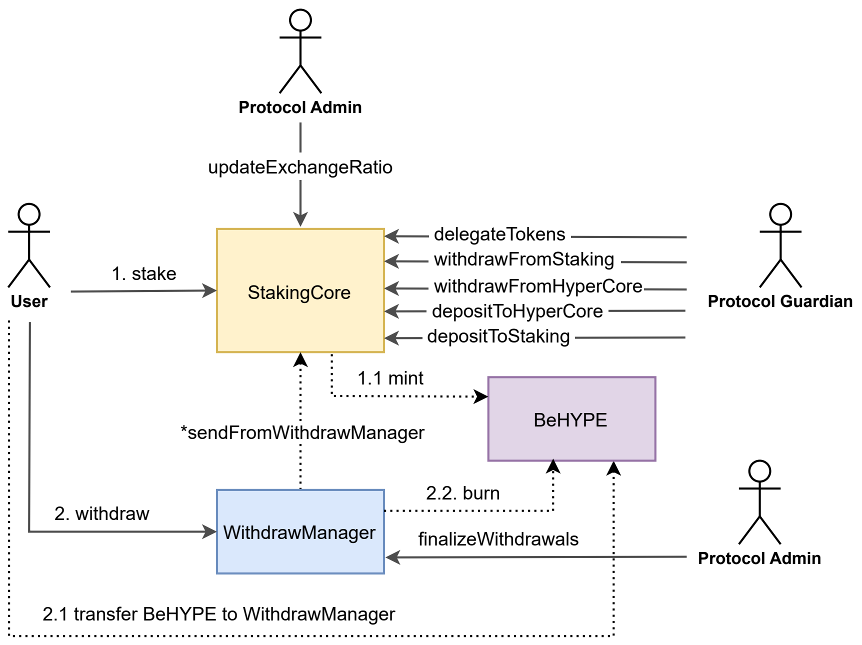


Fig. 2: Communication diagram of core contracts

4.3 beHYPE Token

`beHYPE` is an ERC-20 token that represents staked HYPE. It exposes the standard ERC-20 functions (including `permit`) and is *minted* when users stake HYPE and *burned* when they withdraw. `StakingCore` calls `mint` on `stake` and `WithdrawManager` calls `burn` upon finalizing withdrawals (instant or standard).

4.4 WithdrawManager

The `WithdrawManager` contract handles both instant withdrawals and standard requests (typically finalized within 3–7 days). In summary, it queues standard withdrawals and releases funds only after an off-chain finalization signal.

The `withdraw` function requires the user to transfer **beHYPE** to the contract. It supports **Instant** withdrawals for a fee. The availability for instant withdrawals is determined by checking whether the requested amount does not exceed the withdrawal rate limit and the total amount of beHYPE that can be withdrawn instantly.

The `withdraw` function also allows **Standard** withdrawals: i) a request is recorded and the HYPE amount is fixed at the request-time exchange rate. ii) Once the off-chain service finalizes the withdrawal, the **beHYPE** token amount is burned and the corresponding **HYPE** is transferred to the user.

The function `finalizeWithdrawals(...)` can only be called by `PROTOCOL_ADMIN`. This function processes a batch of withdrawal requests in a loop, sending each HYPE amount to the respective users via `StakingCore`, marking the requests as `finalized`, and then it burns the aggregated `beHYPE` amount.

Roles

a. `PROTOCOL_ADMIN`: finalizes the queue and configures rate limit (capacity/refill).

b. `PROTOCOL_GUARDIAN`: adjusts instant-withdrawal fee (bps).

c. `RoleRegistry`: to pause/unpause withdrawals.

4.5 StakingCore

The `StakingCore` contract maintains the `exchangeRatio` between `beHYPE` and `HYPE` based on the protocol's total balance (HyperEVM + HyperCore), and exposes administrative primitives to interact with HyperCore via `CoreWriter`.

As presented in Fig. 2, the **User** sends `HYPE` when calling the `stake` function and an amount of `beHYPE` token is minted in the current exchange ratio.

Only the `PROTOCOL_ADMIN` can invoke `updateExchangeRatio()` to update the exchange rate between `beHYPE` and `HYPE` by computing the total protocol balance that consists of i) StakingCore contract balance, ii) StakingCore L1 stake account balance and iii) StakingCore L1 spot balance.

Position management on HyperCore: There are five functions that the `PROTOCOL_ADMIN` interacts with HyperCore (Fig. 2). Admins use `deposit/withdraw` *spot*, `deposit/withdraw` *staking* (IDs 4,5), and `delegateTokens` (ID 3) to rebalance between accounts and validators on Core.

The `getTotalProtocolHype()` function sums (converting 8→18 decimals):

a. *Core staking*: `delegated + undelegated + totalPendingWithdrawal` from `delegatorSummary` (precompile).

b. *Core spot*: `spotBalance.total` (precompile).

c. *EVM liquidity*: `address(this).balance`.

4.6 Access Control System (RoleRegistry)

The `RoleRegistry` contract centralizes roles and checks:

a. `PROTOCOL_ADMIN`: critical operations such as, `updateExchangeRatio`, Core actions, queue finalization, rate-limit parameters.

b. `PROTOCOL_GUARDIAN`: guard-rail and fee parameters (acceptable APR, `exchangeRateGuard`, instant fee).

c. `onlyProtocolUpgrader`: used to call `setWithdrawManager` and `_authorizeUpgrade`).

d. *Pause*: the `RoleRegistry` address itself calls `pause/unpause` on `StakingCore` and `WithdrawManager` for incident response.

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] Malicious user can permanently block withdrawal finalizations

File(s): src/WithdrawManager.sol, src/StakingCore.sol

Description: The protocol uses a two-step process for regular withdrawals. First, a user queues a withdrawal request by calling withdraw(...) in the WithdrawManager contract. Later, a privileged PROTOCOL_ADMIN calls finalizeWithdrawals(...) to process a batch of these requests in a loop.

Inside the finalizeWithdrawals(...) loop, the contract iterates through pending entries in the withdrawalQueue and calls the sendFromWithdrawManager(...) function on the StakingCore contract to transfer the HYPE assets to the user.

```
function finalizeWithdrawals(uint256 index) external {
    // ...
    for (uint256 i = lastFinalizedIndex + 1; i <= index;) {
        // @audit The external call to send funds is made for each user in the queue.
        stakingCore.sendFromWithdrawManager(
            withdrawalQueue[i].hypeAmount, withdrawalQueue[i].user
        );
        // ...
        withdrawalQueue[i].finalized = true;

        unchecked { ++i; }
    }
    // ...
}
```

The sendFromWithdrawManager(...) function in StakingCore contains a strict check that requires the underlying HYPE transfer to succeed. If the transfer fails for any reason, the function reverts.

```
function sendFromWithdrawManager(uint256 amount, address to) external {
    if (msg.sender != withdrawManager) revert NotAuthorized();
    (bool success,) = payable(to).call{value: amount}("");
    // @audit-issue If the recipient reverts, this check will fail and cause the entire
    // finalization transaction to revert.
    if (!success) revert FailedToSendFromWithdrawManager();
}
```

A malicious actor can exploit this design to permanently block the withdrawal process. The attacker can deploy a smart contract with a receive() or fallback() function that is designed to always revert upon receiving HYPE. They can then call withdraw(...) to queue a withdrawal request to this malicious contract's address.

When the PROTOCOL_ADMIN attempts to finalize the withdrawals, the loop in finalizeWithdrawals(...) will eventually reach the attacker's entry. The call to sendFromWithdrawManager(...) will fail because the attacker's contract reverts the payment. This failure causes the entire finalizeWithdrawals(...) transaction to revert.

Because the loop always processes withdrawals in sequential order and cannot skip a failing entry, the queue is now permanently stuck at the attacker's position. No user who queued a withdrawal after the attacker can have their withdrawal finalized. This results in a permanent Denial of Service, freezing user funds in the withdrawal queue until a contract upgrade can be deployed to resolve the issue.

Recommendation(s): Consider refactoring the withdrawal mechanism from a "push" to a "pull" pattern.

Status: Fixed.

Update from the client: Fixed in PR #1.

6.2 [Low] Donations to StakingCore can lead to a temporary Denial of Service of updateExchangeRatio(...)

File(s): src/StakingCore.sol

Description: The updateExchangeRatio(...) function in the StakingCore contract is responsible for updating the exchange rate between HYPE and beHYPE. To prevent drastic and potentially erroneous changes, the function includes a safety mechanism, exchangeRateGuard, which ensures that the annual percentage rate (APR) change implied by the new ratio does not exceed a configured acceptablAprInBps threshold.

The calculation of the new exchange ratio relies on the getTotalProtocolHype() function. This function determines the total value locked in the protocol by summing up assets from various sources, including the contract's own HYPE balance (address(this).balance).

```
function getTotalProtocolHype() public view returns (uint256) {
    L1Read.DelegatorSummary memory delegatorSummary = l1Read.delegatorSummary(address(this));
    uint256 totalHypeInStakingAccount = _convertTo18Decimals(delegatorSummary.delegated)
        + _convertTo18Decimals(delegatorSummary.undelegated)
        + _convertTo18Decimals(delegatorSummary.totalPendingWithdrawal);

    L1Read.SpotBalance memory spotBalance = l1Read.spotBalance(address(this), HYPE_TOKEN_ID);
    uint256 totalHypeInSpotAccount = _convertTo18Decimals(spotBalance.total);
    // @audit-issue The contract's liquid balance is part of the total hype calculation.
    uint256 totalHypeInLiquidityPool = address(this).balance;

    uint256 total = totalHypeInStakingAccount + totalHypeInSpotAccount + totalHypeInLiquidityPool;

    return total;
}
```

The problem is that malicious user can donate HYPE directly to the StakingCore contract. This donation artificially inflates the value returned by getTotalProtocolHype() without a corresponding increase in the beHypeToken total supply. This manipulation leads to an inflated newRatio calculation within updateExchangeRatio(...).

```
function updateExchangeRatio() external {
    // ...
    // @audit An inflated totalProtocolHype leads to a larger newRatio.
    uint256 totalProtocolHype = getTotalProtocolHype();

    uint256 newRatio = Math.mulDiv(totalProtocolHype, 1e18, beHypeToken.totalSupply());
    // ...
    uint16 yearlyRateInBps16 = uint16(Math.min(yearlyRateInBps, type(uint16).max));
    if (exchangeRateGuard) {
        // @audit-issue This check can be triggered by an attacker's donation, causing a revert.
        if (yearlyRateInBps16 > acceptablAprInBps) {
            revert ExchangeRatioChangeExceedsThreshold(yearlyRateInBps16);
        }
    }
    // ...
}
```

An attacker can craft the donation amount precisely so that the calculated yearlyRateInBps just exceeds the acceptablAprInBps threshold. As a result, any subsequent legitimate call to updateExchangeRatio(...) by the PROTOCOL_ADMIN will revert, effectively causing a temporary Denial of Service. This blocks a core protocol function. It is relevant to note that this Denial of Service would only be temporary and can be tackled by PROTOCOL_GUARDIAN actions like disabling the exchangeRateGuard.

Recommendation(s): Consider modifying the updateExchangeRatio(...) logic to make it more resilient to sudden asset inflation. Instead of reverting when the exchangeRateGuard threshold is exceeded, the function could cap the newRatio to the maximum value allowed by the acceptablAprInBps for that period.

Status: Acknowledged.

Update from the client: We are aware of this risk but consider the risk acceptable given that the DoS is temporary and can be resolved by the PROTOCOL_GUARDIAN disabling the exchangeRateGuard through the updateExchangeRateGuard() function. This provides us with a quick mitigation path if an attack occurs, allowing normal protocol operations to resume promptly.

6.3 [Low] Instant withdrawal rate limit is stricter than intended

File(s): src/WithdrawManager.sol

Description: The WithdrawManager contract provides an instant withdrawal feature that allows users to bypass the standard withdrawal queue in exchange for a fee. To protect the protocol's liquidity, this feature is governed by a rate limit, instantWithdrawalLimit, which controls the total amount of HYPE that can be withdrawn over a given period.

However, discrepancy in how the rate limit is applied. The rate limit is checked and consumed based on the user's *gross* withdrawal amount (before the fee is deducted). The actual amount of HYPE liquidity that is sent to the user, and thus leaves the protocol, is the *net* amount after the instant withdrawal fee has been subtracted.

The withdraw(...) function first calculates the gross hypeAmount, and then uses this value for the rate limit check and consumption. Only after the rate limit has been depleted is the fee calculated and the smaller, net hypeWithdrawalAfterFee sent to the user.

```
function withdraw(
    uint256 beHypeAmount, bool instant
) external nonReentrant returns (uint256 withdrawalId) {
    // ...
    uint256 hypeAmount = stakingCore.BeHYPEToHYPE(beHypeAmount);

    if (instant) {
        // @audit The rate limit is checked against the gross hypeAmount, before the fee is deducted.
        if (!_canRateLimiterConsume(hypeAmount)) revert InstantWithdrawalRateLimitExceeded();

        // @audit The rate limit bucket is consumed by the gross amount.
        _updateRateLimit(hypeAmount);

        beHypeToken.transferFrom(msg.sender, address(this), beHypeAmount);

        uint256 instantWithdrawalFee = beHypeAmount.mulDiv(
            instantWithdrawalFeeInBps, BASIS_POINT_SCALE
        );
        // ...
        uint256 beHypeWithdrawalAfterFee = beHypeAmount - instantWithdrawalFee;

        // @audit-issue The actual amount of liquidity withdrawn from the protocol is this smaller, net amount.
        uint256 hypeWithdrawalAfterFee = stakingCore.BeHYPEToHYPE(beHypeWithdrawalAfterFee);

        // ...
        stakingCore.sendFromWithdrawManager(hypeWithdrawalAfterFee, msg.sender);
        // ...
    }
    // ...
}
```

This means the instantWithdrawalLimit bucket is depleted faster than liquidity is actually being withdrawn. For example, if a user withdraws an amount equivalent to 100 HYPE with a 2% fee, the rate limit is reduced by 100, but only 98 HYPE is actually paid out.

Recommendation(s): Consider calculating the net withdrawal amount (after fees) *before* checking and consuming from the rate limiter. The rate limit logic in the withdraw(...) function should be based on hypeWithdrawalAfterFee, as this value accurately reflects the amount of liquidity leaving the protocol.

Status: Acknowledged.

Update from the client: The existing behavior is acceptable for our operations.

6.4 [Low] Just-in-Time liquidity strategy on `updateExchangeRatio()` can take yield from stakers

File(s): `src/StakingCore.sol`, `src/WithdrawManager.sol`

Description: The `updateExchangeRatio()` function in the `StakingCore` contract is called periodically by a `PROTOCOL_ADMIN` to realize staking rewards earned by the protocol's assets. When called, the function recalculates the `exchangeRatio` based on the new, larger total of protocol assets against the existing `beHypeToken` supply. This action effectively distributes the accrued yield to all current `beHypeToken` holders.

Because this reward distribution happens in a single, discrete transaction, it creates an opportunity for an attacker to steal yield from long-term stakers. An attacker can watch for a pending `updateExchangeRatio()` transaction and execute a "sandwich attack". They first **front-run** the admin's transaction by calling `stake(...)` with a large amount of capital. Immediately after the admin's transaction processes and distributes the rewards, the attacker **back-runs** it by calling `withdraw(..., true)` to make an instant withdrawal. This "just-in-time" liquidity allows the attacker to claim a share of rewards that accrued over a long period, despite only having funds in the protocol for a moment.

Note that while this attack is mitigated by the instant withdrawal fee, instant withdrawal limits, frequent calls to `updateExchangeRatio()` (which reduces the size of the reward per call), it remains a viable threat if updates become infrequent and the accrued rewards are large enough to offset the fees and gas costs.

Recommendation(s): Consider implementing measures to prevent this just-in-time liquidity attack.

Status: Acknowledged.

Update from the client: We realize the risk here, but see the instant withdrawal fee and the forgoing of staking required for standard withdrawals as a sufficient deterrent.

6.5 [Low] Precision loss during withdrawals can lead to insufficient funds and failed payouts

File(s): src/StakingCore.sol, src/WithdrawManager.sol

Description: The protocol operates across two distinct layers, HyperEVM and HyperCore, which handle the HYPE asset with different levels of precision. In the HyperEVM, HYPE is an 18-decimal asset. In HyperCore, HYPE is treated as an 8-decimal asset.

When a user queues a withdrawal in `WithdrawManager`, the amount of HYPE they are owed is calculated and stored with 18-decimal precision. This high-precision value is also added to the `hypeRequestedForWithdraw` state variable, which tracks the total pending HYPE for withdrawal.

```
function withdraw(uint256 beHypeAmount, bool instant) external /* ... */ {
    // ...
    // @audit hypeAmount is calculated and stored with 18 decimals of precision.
    uint256 hypeAmount = stakingCore.BeHYPEToHYPE(beHypeAmount);

    if (instant) { /* ... */ } else {
        // ...
        // @audit The 18-decimal value is added to the total requested amount.
        hypeRequestedForWithdraw += hypeAmount;
        withdrawalQueue.push(WithdrawalEntry({ /* ..., */ hypeAmount: hypeAmount, /* ... */ }));
        // ...
    }
}
```

To fulfill these withdrawal requests, a `PROTOCOL_ADMIN` must call `withdrawFromHyperCore(...)` in the `StakingCore` contract to bring funds from HyperCore to HyperEVM. During this process, the 18-decimal `amount` specified is converted to 8 decimals by truncating it (dividing by `1e10`).

```
function _convertTo8Decimals(uint256 amount) internal pure returns (uint64) {
    // @audit-issue Division by 1e10 truncates the value, losing decimal precision.
    uint256 truncatedAmount = amount / 1e10;
    if (truncatedAmount > type(uint64).max) revert AmountExceedsUint64Max();
    return uint64(truncatedAmount);
}

function withdrawFromHyperCore(uint256 amount) external {
    // ...
    // @audit This check prevents rounding up the amount to compensate for truncation loss.
    if (amount > IWithdrawManager(withdrawManager).hypeRequestedForWithdraw()) revert NotAuthorized();

    _encodeAction(6, abi.encode(L1_HYPE_CONTRACT, HYPE_TOKEN_ID, _convertTo8Decimals(amount)));
    // ...
}
```

Because of this truncation, the amount of HYPE brought from HyperCore can be slightly less than the amount needed to fullfil the requests. The check `amount > hypeRequestedForWithdraw` prevents the admin from rounding up the withdrawal amount to compensate for this loss. Consequently, the `StakingCore` contract can end up with insufficient balance to pay all queued users. This may cause the `finalizeWithdrawals(...)` transaction to revert for the last users in a batch.

Recommendation(s): Consider standardizing the precision used for withdrawal accounting to match the lower-precision system (HyperCore).

Status: Acknowledged.

Update from the client: We are comfortable with the risk here as `withdrawFromHyperCore(...)` can be called again if more liquidity is needed for payouts.

6.6 [Info] Queued withdrawal payout amount is fixed at the time of request and does not account for slashing

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

Description: The design of the non-instant withdrawal process involves calculating and storing the final `hypeAmount` payout at the moment a user initiates their request via the `withdraw(...)` function. This calculation is based on the `exchangeRatio` active at that specific block, and the resulting `hypeAmount` is saved within the `WithdrawalEntry` struct.

```
function withdraw(
    uint256 beHypeAmount, bool instant
) external nonReentrant returns (uint256 withdrawalId) {
    // ...
    // @audit The hypeAmount is calculated using the exchangeRatio at the time of request.
    uint256 hypeAmount = stakingCore.BeHYPEToHYPE(beHypeAmount);

    if (instant) {
        // ...
    } else {
        // ...
        withdrawalQueue.push(WithdrawalEntry({
            user: msg.sender,
            beHypeAmount: beHypeAmount,
            // @audit This value is stored and remains fixed until finalization.
            hypeAmount: hypeAmount,
            finalized: false
        }));
        // ...
    }
}
```

This stored value is then used directly during the finalization step when an admin calls `finalizeWithdrawals(...)` to transfer the funds.

```
function finalizeWithdrawals(uint256 index) external {
    // ...
    for (uint256 i = lastFinalizedIndex + 1; i <= index;) {
        // @audit The payout uses the fixed hypeAmount that was stored when the request was made.
        stakingCore.sendFromWithdrawManager(
            withdrawalQueue[i].hypeAmount, withdrawalQueue[i].user
        );
        // ...
    }
    // ...
}
```

An important consequence of this design is that the payout amount is locked in and does not adjust to fluctuations in the `exchangeRatio` that may occur during the withdrawal delay period. If a slashing event were to take place on the L1—causing a reduction in the protocol's total assets and a corresponding decrease in the `exchangeRatio`—a pending withdrawal would still pay out the higher, pre-slashing amount.

Recommendation(s): The team should be aware that the current implementation fixes the withdrawal amount at the time of the request. Consider if this behavior is intended, especially in the context of potential slashing events. An alternative design would be to calculate the final `hypeAmount` at the moment of finalization and use the lowest of the two values.

Status: Acknowledged.

Update from the client: Hyperliquid has no automated slashing implemented, making this issue highly unlikely.

6.7 [Best Practice] `finalizeWithdrawals()` violates the Checks-Effects-Interactions pattern

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

Description: The Checks-Effects-Interactions (CEI) pattern specifies that contracts should perform all state-modifying actions (Effects) *before* making any external calls (Interactions). Adhering to this pattern prevents re-entrancy attacks, where an untrusted external contract can hijack the execution flow and call back into the original contract while it is in an inconsistent state.

The `finalizeWithdrawals()` function in the `WithdrawManager` contract violates this pattern. Inside its processing loop, it makes an external call to `stakingCore.sendFromWithdrawManager(...)` to transfer funds *before* updating part of the state, as for example, marking the withdrawal as finalized.

This `Interaction -> Effect` ordering is dangerous. It transfers execution control to the withdrawal recipient's address while the `WithdrawManager` contract is in an intermediate state. During this window, the contract's state would be inconsistent. For example, the attacker's withdrawal would still be marked as `finalized = false` even though the funds have already been sent. While the `nonReentrant` modifier on the `withdraw(...)` function prevents some obvious exploits, other view functions like `getUserUnFinalizedWithdrawals(...)` could return stale data.

Recommendation(s): Consider refactoring the `finalizeWithdrawals(...)` function to strictly adhere to the Checks-Effects-Interactions pattern.

Status: Fixed.

Update from the client: Fixed in PR #1.

7 Documentation Evaluation

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

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

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

Remarks about the Hyperbeat documentation

The documentation for the Hyperbeat protocol was provided through docs.hyperbeat and README file with a high-level description and Hyperliquid GitBook. Moreover, the Hyperbeat team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Compilation Output

```
# forge build
[] Compiling...
[] Compiling 89 files with Solc 0.8.30
[] Solc 0.8.30 finished in 11.96s
Compiler run successful!
note[mixed-case-function]: function names should use mixedCase
  --> /hyperbeat/src/RoleRegistry.sol:27:14
   |
27 |     function MAX_ROLE() public pure returns (uint256) {
   |              --------
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-function

note[unaliased-plain-import]: use named imports '{A, B}' or alias 'import ".." as X'
 --> /hyperbeat/src/BeHYPETimelock.sol:4:8
  |
4 | import "@openzeppelin/contracts/governance/TimelockController.sol";
  |        -----------------------------------------------------------
  = help: https://book.getfoundry.sh/reference/forge/forge-lint#unaliased-plain-import

note[unaliased-plain-import]: use named imports '{A, B}' or alias 'import ".." as X'
 --> /hyperbeat/src/interfaces/IBeHYPE.sol:4:8
  |
4 | import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
  |        ------------------------------------------------
  = help: https://book.getfoundry.sh/reference/forge/forge-lint#unaliased-plain-import

note[mixed-case-variable]: mutable variables should use mixedCase
   --> /hyperbeat/src/lib/L1Read.sol:238:12
    |
238 |     uint32 perp_dex_index,
    |            --------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-variable

note[unaliased-plain-import]: use named imports '{A, B}' or alias 'import ".." as X'
  --> /hyperbeat/test/Base.t.sol:20:8
   |
20 | import "forge-std/console.sol";
   |        -----------------------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#unaliased-plain-import

note[mixed-case-variable]: mutable variables should use mixedCase
  --> /hyperbeat/test/Base.t.sol:25:19
   |
25 |     BeHYPE public beHYPE;
   |                   ------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-variable

note[unaliased-plain-import]: use named imports '{A, B}' or alias 'import ".." as X'
 --> /hyperbeat/test/BeHYPE.t.sol:4:8
  |
4 | import "./Base.t.sol";
  |        --------------
  |
  = help: https://book.getfoundry.sh/reference/forge/forge-lint#unaliased-plain-import

note[unaliased-plain-import]: use named imports '{A, B}' or alias 'import ".." as X'
 --> /hyperbeat/test/BeHYPE.t.sol:5:8
  |
5 | import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
  |        ------------------------------------------------------
  |
  = help: https://book.getfoundry.sh/reference/forge/forge-lint#unaliased-plain-import
```

```
note[mixed-case-variable]: mutable variables should use mixedCase
  --> /hyperbeat/test/Base.t.sol:63:16
   |
63 |         BeHYPE beHYPEImpl = new BeHYPE();
   |                ----------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-variable

note[unused-import]: unused imports should be removed
  --> /hyperbeat/test/Base.t.sol:13:9
   |
13 | import {IStakingCore} from "../src/interfaces/IStakingCore.sol";
   |         ------------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#unused-import

note[unused-import]: unused imports should be removed
  --> /hyperbeat/test/Base.t.sol:14:9
   |
14 | import {IWithdrawManager} from "../src/interfaces/IWithdrawManager.sol";
   |         ----------------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#unused-import

note[unaliased-plain-import]: use named imports '{A, B}' or alias 'import ".." as X'
  --> /hyperbeat/src/WithdrawManager.sol:17:8
   |
17 | import "forge-std/console.sol";
   |        -----------------------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#unaliased-plain-import

warning[erc20-unchecked-transfer]: ERC20 'transfer' and 'transferFrom' calls should check the return value
   --> /hyperbeat/src/WithdrawManager.sol:113:13
    |
113 |             beHypeToken.transferFrom(msg.sender, address(this), beHypeAmount);
    |             -----------------------------------------------------------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#erc20-unchecked-transfer

warning[erc20-unchecked-transfer]: ERC20 'transfer' and 'transferFrom' calls should check the return value
   --> /hyperbeat/src/WithdrawManager.sol:116:13
    |
116 |             beHypeToken.transfer(roleRegistry.protocolTreasury(), instantWithdrawalFee);
    |             ---------------------------------------------------------------------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#erc20-unchecked-transfer

warning[erc20-unchecked-transfer]: ERC20 'transfer' and 'transferFrom' calls should check the return value
   --> /hyperbeat/src/WithdrawManager.sol:129:13
    |
129 |             beHypeToken.transferFrom(msg.sender, address(this), beHypeAmount);
    |             -----------------------------------------------------------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#erc20-unchecked-transfer

note[mixed-case-function]: function names should use mixedCase
   --> /hyperbeat/src/WithdrawManager.sol:232:14
    |
232 |     function getTotalInstantWithdrawableBeHYPE() public view returns (uint256) {
    |              ---------------------------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-function

note[mixed-case-function]: function names should use mixedCase
   --> /hyperbeat/src/WithdrawManager.sol:257:14
    |
257 |     function lowWatermarkInHYPE() public view returns (uint256) {
    |              ------------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-function
```

```
note[mixed-case-function]: function names should use mixedCase
  --> /hyperbeat/src/interfaces/IRoleRegistry.sol:59:14
   |
59 |     function MAX_ROLE() external pure returns (uint256);
   |              --------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-function

note[mixed-case-function]: function names should use mixedCase
   --> /hyperbeat/src/interfaces/IRoleRegistry.sol:121:14
    |
121 |     function PROTOCOL_PAUSER() external view returns (bytes32);
    |              ---------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-function

note[mixed-case-function]: function names should use mixedCase
   --> /hyperbeat/src/interfaces/IRoleRegistry.sol:128:14
    |
128 |     function PROTOCOL_ADMIN() external view returns (bytes32);
    |              --------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-function

note[mixed-case-function]: function names should use mixedCase
   --> /hyperbeat/src/interfaces/IRoleRegistry.sol:135:14
    |
135 |     function PROTOCOL_GUARDIAN() external view returns (bytes32);
    |              -----------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-function

note[unused-import]: unused imports should be removed
  --> /hyperbeat/test/RoleRegistry.t.sol:10:9
   |
10 | import {Ownable2StepUpgradeable} from "@openzeppelin-upgradeable/access/Ownable2StepUpgradeable.sol";
   |         -----------------------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#unused-import

note[unaliased-plain-import]: use named imports '{A, B}' or alias 'import ".." as X'
 --> /hyperbeat/test/WithdrawManager.t.sol:4:8
  |
4 | import "./Base.t.sol";
  |        --------------
  |
  = help: https://book.getfoundry.sh/reference/forge/forge-lint#unaliased-plain-import

note[unused-import]: unused imports should be removed
  --> /hyperbeat/test/Base.t.sol:15:9
   |
15 | import {IRoleRegistry} from "../src/interfaces/IRoleRegistry.sol";
   |         -------------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#unused-import

note[mixed-case-variable]: mutable variables should use mixedCase
  --> /hyperbeat/test/WithdrawManager.t.sol:52:17
   |
52 |         uint256 balanceBeforeBeHYPE = beHYPE.balanceOf(address(withdrawManager));
   |                 -------------------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-variable

note[mixed-case-function]: function names should use mixedCase
   --> /hyperbeat/src/interfaces/IStakingCore.sol:196:14
    |
196 |     function BeHYPEToHYPE(uint256 kHYPEAmount) external view returns (uint256);
    |              ------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-function
```

```
note[mixed-case-variable]: mutable variables should use mixedCase
   --> /hyperbeat/src/interfaces/IStakingCore.sol:196:35
    |
196 |     function BeHYPEToHYPE(uint256 kHYPEAmount) external view returns (uint256);
    |                                   -----------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-variable

note[mixed-case-function]: function names should use mixedCase
   --> /hyperbeat/src/interfaces/IStakingCore.sol:204:14
    |
204 |     function HYPEToBeHYPE(uint256 HYPEAmount) external view returns (uint256);
    |              ------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-function

note[unaliased-plain-import]: use named imports '{A, B}' or alias 'import ".." as X'
  --> /hyperbeat/src/StakingCore.sol:16:8
   |
16 | import "forge-std/console.sol";
   |        -----------------------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#unaliased-plain-import

note[screaming-snake-case-const]: constants should use SCREAMING_SNAKE_CASE
  --> /hyperbeat/src/StakingCore.sol:34:28
   |
34 |     L1Read public constant l1Read = L1Read(0xb7467E0524Afba7006957701d1F06A59000d15A2);
   |                            ------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#screaming-snake-case-const

note[unaliased-plain-import]: use named imports '{A, B}' or alias 'import ".." as X'
 --> /hyperbeat/test/StakingCore.t.sol:4:8
  |
4 | import "./Base.t.sol";
  |        --------------
  |
  = help: https://book.getfoundry.sh/reference/forge/forge-lint#unaliased-plain-import

warning[erc20-unchecked-transfer]: ERC20 'transfer' and 'transferFrom' calls should check the return value
   --> /hyperbeat/test/BeHYPE.t.sol:134:9
    |
134 |         beHYPE.transfer(user2, 50 ether);
    |         --------------------------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#erc20-unchecked-transfer

warning[erc20-unchecked-transfer]: ERC20 'transfer' and 'transferFrom' calls should check the return value
   --> /hyperbeat/test/BeHYPE.t.sol:158:9
    |
158 |         beHYPE.transferFrom(user, user2, 50 ether);
    |         ------------------------------------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#erc20-unchecked-transfer

note[screaming-snake-case-const]: constants should use SCREAMING_SNAKE_CASE
  --> /hyperbeat/src/StakingCore.sol:35:32
   |
35 |     CoreWriter public constant coreWriter = CoreWriter(0x3333333333333333333333333333333333333333);
   |                                ----------
   |
   = help: https://book.getfoundry.sh/reference/forge/forge-lint#screaming-snake-case-const

note[mixed-case-variable]: mutable variables should use mixedCase
   --> /hyperbeat/src/interfaces/IStakingCore.sol:204:35
    |
204 |     function HYPEToBeHYPE(uint256 HYPEAmount) external view returns (uint256);
    |                                   ----------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-variable
```

```
note[unused-import]: unused imports should be removed
 --> /hyperbeat/src/interfaces/IStakingCore.sol:4:9
  |
4 | import {IRoleRegistry} from "./IRoleRegistry.sol";
  |         -------------
  |
  = help: https://book.getfoundry.sh/reference/forge/forge-lint#unused-import

note[unused-import]: unused imports should be removed
 --> /hyperbeat/src/interfaces/IStakingCore.sol:5:9
  |
5 | import {IBeHYPEToken} from "./IBeHype.sol";
  |         ------------
  |
  = help: https://book.getfoundry.sh/reference/forge/forge-lint#unused-import

note[mixed-case-variable]: mutable variables should use mixedCase
   --> /hyperbeat/test/BeHYPE.t.sol:170:16
    |
170 |         BeHYPE newBeHYPEImpl = new BeHYPE();
    |                -------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-variable

note[mixed-case-function]: function names should use mixedCase
   --> /hyperbeat/src/StakingCore.sol:192:14
    |
192 |     function BeHYPEToHYPE(uint256 kHYPEAmount) public view returns (uint256) {
    |              ------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-function

note[mixed-case-variable]: mutable variables should use mixedCase
   --> /hyperbeat/src/StakingCore.sol:192:35
    |
192 |     function BeHYPEToHYPE(uint256 kHYPEAmount) public view returns (uint256) {
    |                                   -----------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-variable

note[unused-import]: unused imports should be removed
 --> /hyperbeat/src/interfaces/IWithdrawManager.sol:4:9
  |
4 | import {IRoleRegistry} from "./IRoleRegistry.sol";
  |         -------------
  |
  = help: https://book.getfoundry.sh/reference/forge/forge-lint#unused-import

note[unused-import]: unused imports should be removed
 --> /hyperbeat/src/interfaces/IWithdrawManager.sol:5:9
  |
5 | import {IBeHYPEToken} from "./IBeHYPE.sol";
  |         ------------
  |
  = help: https://book.getfoundry.sh/reference/forge/forge-lint#unused-import

note[unused-import]: unused imports should be removed
 --> /hyperbeat/src/interfaces/IWithdrawManager.sol:6:9
  |
6 | import {IStakingCore} from "./IStakingCore.sol";
  |         ------------
  |
  = help: https://book.getfoundry.sh/reference/forge/forge-lint#unused-import

note[mixed-case-variable]: mutable variables should use mixedCase
   --> /hyperbeat/test/BeHYPE.t.sol:190:16
    |
190 |         BeHYPE newBeHYPEImpl = new BeHYPE();
    |                -------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-variable
```

```
note[mixed-case-function]: function names should use mixedCase
   --> /hyperbeat/src/StakingCore.sol:196:14
    |
196 |     function HYPEToBeHYPE(uint256 HYPEAmount) public view returns (uint256) {
    |              ------------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-function

note[mixed-case-variable]: mutable variables should use mixedCase
   --> /hyperbeat/src/StakingCore.sol:196:35
    |
196 |     function HYPEToBeHYPE(uint256 HYPEAmount) public view returns (uint256) {
    |                                   ----------
    |
    = help: https://book.getfoundry.sh/reference/forge/forge-lint#mixed-case-variable
```

8.2 Tests Output

```
# forge test
[] Compiling...
[] Compiling 6 files with Solc 0.8.30
[] Solc 0.8.30 finished in 10.13s
Compiler run successful!

Ran 13 tests for test/BeHYPE.t.sol:BeHYPETest
[PASS] test_Approve() (gas: 96773)
[PASS] test_Burn() (gas: 80917)
[PASS] test_InitialState() (gas: 37065)
[PASS] test_Mint() (gas: 71662)
[PASS] test_Permit() (gas: 80616)
[PASS] test_RevertPermitExpired() (gas: 28714)
[PASS] test_RevertReinitialization() (gas: 26179)
[PASS] test_RevertUnauthorizedBurn() (gas: 71986)
[PASS] test_RevertUnauthorizedMint() (gas: 19160)
[PASS] test_RevertUpgradeUnauthorized() (gas: 1738540)
[PASS] test_Transfer() (gas: 98323)
[PASS] test_TransferFrom() (gas: 107258)
[PASS] test_UpgradeProxy() (gas: 1837896)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 3.29ms (2.04ms CPU time)

Ran 10 tests for test/WithdrawManager.t.sol:WithdrawManagerTest
[PASS] test_RevertReinitialization() (gas: 26178)
[PASS] test_RevertUpgradeUnauthorized() (gas: 2111069)
[PASS] test_UpgradeSuccess() (gas: 2121584)
[PASS] test_instantWithdrawal() (gas: 369689)
[PASS] test_instantWithdrawalRateLimit() (gas: 357375)
[PASS] test_instantWithdrawal_with_exchange_rate() (gas: 396874)
[PASS] test_multipleUsersMultipleWithdrawals() (gas: 1745779)
[PASS] test_withdraw() (gas: 481488)
[PASS] test_withdraw_finalization_reverts() (gas: 1254916)
[PASS] test_withdraw_with_exchange_rate() (gas: 924823)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 3.03ms (3.73ms CPU time)

Ran 11 tests for test/RoleRegistry.t.sol:RoleRegistryTest
[PASS] test_RevertReinitialization() (gas: 26029)
[PASS] test_RevertUpgradeUnauthorized() (gas: 1173720)
[PASS] test_TwoStepOwnershipTransfer() (gas: 103264)
[PASS] test_UpgradeSuccess() (gas: 1183841)
[PASS] test_pauseAndUnpauseProtocol_cycle() (gas: 388277)
[PASS] test_pauseProtocol() (gas: 112566)
[PASS] test_pauseProtocol_revertIfNotPauser() (gas: 19509)
[PASS] test_setProtocolTreasury() (gas: 27591)
[PASS] test_setProtocolTreasury_revertIfNotGuardian() (gas: 20676)
[PASS] test_unpauseProtocol() (gas: 325996)
[PASS] test_unpauseProtocol_revertIfNotUnpauser() (gas: 97785)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 3.27ms (1.35ms CPU time)

Ran 9 tests for test/StakingCore.t.sol:StakingCoreTest
[PASS] test_ExchangeRatio() (gas: 187157)
[PASS] test_ExchangeRatio_negative() (gas: 187576)
[PASS] test_ExchangeRatio_should_revert_if_apr_exceeds_threshold() (gas: 179041)
[PASS] test_ExchangeRatio_should_revert_if_apr_exceeds_threshold_negative() (gas: 185191)
[PASS] test_MockDepositToHyperCore() (gas: 224101)
[PASS] test_RevertReinitialization() (gas: 25932)
[PASS] test_RevertUpgradeUnauthorized() (gas: 1975565)
[PASS] test_UpgradeSuccess() (gas: 1983701)
[PASS] test_stake() (gas: 141814)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 3.20ms (1.64ms CPU time)

Ran 4 test suites in 7.53ms (12.79ms CPU time): 43 tests passed, 0 failed, 0 skipped (43 total tests)
```

Coverage Analysis

In summary, overall coverage is solid 86.05% lines, 82.51% statements, 78.79% functions, but only 44.23% branches; BeHYPE.sol is fully covered across all metrics. The weakest spots are branch coverage in `StakingCore` (36.36%) and `WithdrawManager` (42.86%), indicating untested conditionals.

```
------------------------------------+------------------+------------------+----------------+-----------------
| File                               | % Lines          | % Statements     | % Branches     | % Funcs         |
+==============================================================================================================+
| src/BeHYPE.sol                     | 100.00% (25/25)  | 100.00% (20/20)  | 100.00% (2/2)  | 100.00% (7/7)   |
|------------------------------------+------------------+------------------+----------------+-----------------|
| src/RoleRegistry.sol               | 81.63% (40/49)   | 79.07% (34/43)   | 57.14% (4/7)   | 75.00% (12/16)  |
|------------------------------------+------------------+------------------+----------------+-----------------|
| src/StakingCore.sol                | 86.11% (93/108)  | 80.92% (106/131) | 36.36% (8/22)  | 81.82% (18/22)  |
|------------------------------------+------------------+------------------+----------------+-----------------|
| src/WithdrawManager.sol            | 84.82% (95/112)  | 82.58% (109/132) | 42.86% (9/21)  | 71.43% (15/21)  |
|------------------------------------+------------------+------------------+----------------+-----------------|
| Total                              | 86.05% (253/294) | 82.51% (269/326) | 44.23% (23/52) | 78.79% (52/66) |
------------------------------------+------------------+------------------+----------------+-----------------
```

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