Pashov Audit Group

Hyperbeat Security Review

September 3rd 2025 - September 5th 2025

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Risk Classification 3
4. About Hyperbeat 4
5. Executive Summary 4
6. Findings 5
Medium findings 7
[M-01] `getTotalProtocolHype()` includes pending withdrawal amounts 7
[M-02] Idle HYPE balance consideration allows silent bypass of exchange rate guard 8
Low findings 10
[L-01] `_convertToBucketUnit()` uses `<` instead of `<=` to check for overflow 10
[L-02] Event spamming because `stake()` does not enforce a minimum amount 10
[L-03] `BeHYPE` and `WithdrawManager` contracts incorrectly import `IBeHYPE.sol` 10
[L-04] Precision loss in `updateExchangeRatio()` 10
[L-05] Treasury receives staking rewards due to fees being taken in form of beHYPE 11
[L-06] `emergencyWithdrawFromStaking()` does not update `lastWithdrawalTimestamp` 11
[L-07] Rounding down in fee calculation causes protocol revenue loss over time 12
[L-08] Withdraw from staking limit can be bypassed via multiple calls 12
[L-09] Funds not transferred before `withdrawManager` update 12
[L-10] `depositToHyperCore()` fails to account for `hypeRequestedForWithdraw` 13
[L-11] First staker uses wrong `exchangeRatio` if `totalSupply()` of beHYPE is 0 14
[L-12] Loss events may be blocked from updating exchange ratio due to APR guard 16
[L-13] Rounding in `stake()` is in favour of users 16
[L-14] Pending withdrawals are not considered in total Hype assets 17
[L-15] `StakingCore` does not account for pending deposits to Hyper Core 18

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Hyperbeat

Hyperbeat is a staking solution where users deposit HYPE in exchange for beHYPE tokens with instant or delayed withdrawals, and dynamic exchange rate management.

5. Executive Summary

A time-boxed security review of the **etherfi-protocol/BeHYPE** repository was done by Pashov Audit Group, during which **IvanFitro**, **0xl33**, **unforgiven**, **Tejas Warambhe** engaged to review **Hyperbeat**. A total of **17** issues were uncovered.

Protocol Summary	
Project Name	Hyperbeat
Protocol Type	Staked Token
Timeline	September 3rd 2025 - September 5th 2025

Review commit hash:

- c620e61aaf8fe5767e713bb9d7447bd9fbc40148 (etherfi-protocol/BeHYPE)

Scope

`BeHYPE.sol` `BeHYPETimelock.sol` `RoleRegistry.sol` `StakingCore.sol` `WithdrawManager.sol` `interfaces/`

6. Findings

Findings count

Severity	Amount
Medium	2
Low	15
Total findings	**17**

Summary of findings

ID	Title	Severity	Status
[M-01]	`getTotalProtocolHype()` includes pending withdrawal amounts	Medium	Acknowledged
[M-02]	Idle HYPE balance consideration allows silent bypass of exchange rate guard	Medium	Acknowledged
[L-01]	`_convertToBucketUnit()` uses `<` instead of `<=` to check for overflow	Low	Acknowledged
[L-02]	Event spamming because `stake()` does not enforce a minimum amount	Low	Acknowledged
[L-03]	`BeHYPE` and `WithdrawManager` contracts incorrectly import `IBeHYPE.sol`	Low	Acknowledged
[L-04]	Precision loss in `updateExchangeRatio()`	Low	Acknowledged
[L-05]	Treasury receives staking rewards due to fees being taken in form of beHYPE	Low	Acknowledged
[L-06]	`emergencyWithdrawFromStaking()` does not update `lastWithdrawalTimestamp`	Low	Acknowledged
[L-07]	Rounding down in fee calculation causes protocol revenue loss over time	Low	Acknowledged
[L-08]	Withdraw from staking limit can be bypassed via multiple calls	Low	Acknowledged
[L-09]	Funds not transferred before `withdrawManager` update	Low	Acknowledged
[L-10]	`depositToHyperCore()` fails to account for `hypeRequestedForWithdraw`	Low	Acknowledged

ID	Title	Severity	Status
[L-11]	First staker uses wrong `exchangeRatio` if `totalSupply()` of beHYPE is 0	Low	Acknowledged
[L-12]	Loss events may be blocked from updating exchange ratio due to APR guard	Low	Acknowledged
[L-13]	Rounding in `stake()` is in favour of users	Low	Acknowledged
[L-14]	Pending withdrawals are not considered in total Hype assets	Low	Acknowledged
[L-15]	`StakingCore` does not account for pending deposits to Hyper Core	Low	Acknowledged

Medium findings

[M-01] `getTotalProtocolHype()` includes pending withdrawal amounts

Severity

Impact: Medium

Likelihood: Medium

Description

The `getTotalProtocolHype()` function in `StakingCore.sol` calculates the total HYPE owned by the protocol, but fails to account for HYPE that has already been requested for withdrawal but not yet finalized. This leads to inflated TVL calculations when used during instant withdrawal flow.

When users queue withdrawals via `WithdrawManager.withdraw()`, the `hypeRequestedForWithdraw` amount increases, but `getTotalProtocolHype()` still includes this HYPE in its calculation. This creates a window where users can perform instant withdrawals beyond what should be allowed.

The issue occurs because: 1. `getTotalProtocolHype()` returns the full balance without subtracting the pending withdrawals. 2. `getLiquidHypeAmount()` returns the full balance without subtracting the pending withdrawals. 3. `getTotalInstantWithdrawableBeHYPE()` uses these inflated values to calculate the `withdrawableAmount`. 4. This allows more instant withdrawals than the actual available liquidity should permit.

The bug only manifests when there are unfinalized withdrawals in the queue. Once `finalizeWithdrawals()` is called, the HYPE is transferred from `StakingCore` to `WithdrawManager`, naturally reducing the returned values.

Note: `getTotalProtocolHype()` should only subtract `hypeRequestedForWithdraw` when called during `withdraw()` flow, while preserving the current behavior when called during `updateExchangeRatio()` flow. This is because subtracting the pending withdrawals during `updateExchangeRatio()` flow would result in an incorrect exchange rate calculation.

Consider this scenario:

1. Protocol has `10000e18` HYPE total.
2. Low watermark is 10% (`1000e18`) HYPE.
3. User queues withdrawal of `1000e18` HYPE.
4. Admin undelegates, transfers from staking to spot balance in HyperCore, withdraws from HyperCore to StakingCore contract.

5. `getTotalProtocolHype()` still returns `10000e18` HYPE (`finalizeWithdrawals()` not called yet).
6. `lowWatermarkInHYPE()` calculates as `1000e18` HYPE (10% of `10000e18`).
7. `getLiquidHypeAmount()` returns `2000e18` HYPE (`1000e18` is left there by default + `1000e18` withdrawn from HyperCore to fulfill the withdrawal request).
8. Available for instant withdrawal: `withdrawableAmount = 2000e18 - 1000e18 = 1000e18` HYPE <--- this is wrong.
9. Actually, it should be available: `withdrawableAmount = 1000e18 - 900e18 = 100e18` HYPE, because `1000e18` HYPE that is included in the total does not belong to the protocol.
10. Before `finalizeWithdrawals()` is called, user can instantly withdraw `1000e18` HYPE instead of the correct `100e18` HYPE limit.
11. Admin calls `finalizeWithdrawals()` and balance of `StakingCore` becomes 0, even though at this point it should be `900e18` (10% of `9000e18`).

Recommendations

1. Create a separate function `getAvailableProtocolHype()` that does the same thing as `getTotalProtocolHype()` but subtracts `hypeRequestedForWithdraw` from total. Use it in `lowWatermarkInHYPE()` function.
2. In `getLiquidHypeAmount()` , subtract `hypeRequestedForWithdraw` from the balance (make sure underflow is handled):

```
function getLiquidHypeAmount() public view returns (uint256) {
    if (address(stakingCore).balance <= hypeRequestedForWithdraw) return 0;
    return address(stakingCore).balance - hypeRequestedForWithdraw;
}
```

After these mitigations, the instant withdrawal limit will be correctly enforced both before and after `finalizeWithdrawals()` call.

[M-02] Idle HYPE balance consideration allows silent bypass of exchange rate guard

Severity

Impact: Medium

Likelihood: Medium

Description

The `updateExchangeRatio()` function utilizes an exchange rate guard to mitigate potential issues with bad reads from precompiles in HyperCore. The new ratio is calculated as per the protocol's token holdings:

```
    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);

        uint256 totalHypeInLiquidityPool = address(this).balance;
<<@

        uint256 total = totalHypeInStakingAccount + totalHypeInSpotAccount +
totalHypeInLiquidityPool;

        return total;
    }
```

However, the guard acts as a whole on APR, considering the entire balance, including the HYPE value present in the `StakingCore` contract, which is not a part of the tokens present in the HyperCore.

Hence, a bad precompile read can silently bypass the guard due to sufficient idle liquidity in the `StakingCore` contract, corrupting the ratio and affecting `WithdrawManager::lowWatermarkInHYPE()`.

Recommendations

It is recommended to guard the `delegatorSummary`'s values individually.

Low findings

[L-01] `_convertToBucketUnit()` uses `<` instead of `<=` to check for overflow

`_convertToBucketUnit()` uses `amount < type(uint64).max * BUCKET_UNIT_SCALE` to check if the result would overflow `uint64`. The correct approach is to use `<=` for the comparison, because otherwise a valid value is excluded even though it does not overflow.

Recommendation: Replace `<` with `<=`.

[L-02] Event spamming because `stake()` does not enforce a minimum amount

`stake()` allows users to mint beHYPE in exchange for HYPE. The problem is that it does not enforce a minimum staking amount, which can lead to spammed `Deposit` events. This makes it difficult to track meaningful activity off-chain, especially if `communityCode` needs to be tracked.

Recommendation: Set a minimum amount that users must stake.

[L-03] `BeHYPE` and `WithdrawManager` contracts incorrectly import `IBeHYPE.sol`

`BeHYPE` and `WithdrawManager` import the `IBeHYPE` interface, but the contracts are using `IBeHype.sol` instead of `IBeHYPE.sol`, which prevents them from compiling.

Recommendation: Import `IBeHYPE.sol` instead of `IBeHype.sol` to correctly reference the interface.

[L-04] Precision loss in `updateExchangeRatio()`

The `updateExchangeRatio()` function performs sequential mathematical operations that introduce cumulative precision loss, potentially causing the `yearlyRateInBps16` value to be smaller than it should be. The function calculates percentage change, then yearly rate, then converts to basis points through multiple division operations:

```
uint256 percentageChange = Math.mulDiv(ratioChange, 1e18, exchangeRatio);
uint256 yearlyRate = Math.mulDiv(percentageChange, 365 days, elapsedTime);
uint256 yearlyRateInBps = yearlyRate / 1e14;
```

While `Math.mulDiv` preserves precision for individual operations, the sequential approach compounds small rounding errors. The `percentageChange` result (already rounded) is used as input for the `yearlyRate` calculation, and the final division by `1e14` uses regular division, which truncates any fractional part.

This precision loss only affects the `exchangeRateGuard` check that compares `yearlyRateInBps16` against `acceptablAprInBps`. In edge cases where the actual rate is just above the threshold but precision loss reduces the calculated value below it, the guard check could be incorrectly bypassed, allowing exchange ratio updates that should have been rejected.

Consider restructuring the calculation to minimize intermediate rounding by combining operations where possible, ensuring multiplications occur before divisions.

For example:
`yearlyRateInBps = Math.mulDiv(ratioChange * 365 days, 1e18, exchangeRatio * elapsedTime * 1e14)` (overflow should not be possible here).

[L-05] Treasury receives staking rewards due to fees being taken in form of beHYPE

During instant withdrawals, the withdrawal fee is collected in beHYPE tokens and transferred to the protocol treasury. Since beHYPE tokens are share tokens that accrue staking rewards through an increasing exchange rate, the treasury effectively captures a portion of user staking rewards over time. The longer the treasury holds these beHYPE tokens, the more rewards are diverted from users to the treasury, which is unfair to stakers.

Consider transferring the withdrawal fees to the treasury in the form of HYPE tokens, as HYPE tokens do not accrue staking rewards and will not divert rewards away from users.

[L-06] `emergencyWithdrawFromStaking()` does not update `lastWithdrawalTimestamp`

The `emergencyWithdrawFromStaking()` function performs the same withdrawal action as `withdrawFromStaking()` but fails to update the `lastWithdrawalTimestamp` state variable. This inconsistency means that emergency withdrawals do not reset the withdrawal cooldown period, potentially allowing bypassing of the intended cooldown mechanism between regular withdrawals.

Consider updating the `lastWithdrawalTimestamp` state variable in `emergencyWithdrawFromStaking()` to match the behavior of `withdrawFromStaking()`.

[L-07] Rounding down in fee calculation causes protocol revenue loss over time

In the `withdraw()` function for instant withdrawals, the protocol calculates the withdrawal fee as:

```solidity uint256 instantWithdrawalFee = beHypeAmount.mulDiv(instantWithdrawalFeeInBps, BASIS_POINT_SCALE); ````

Because `mulDiv` defaults to rounding down, the calculated fee is always truncated in favor of the user. Although dust withdrawals are prevented by `minWithdrawalAmount`, this rounding effect still causes the protocol to systematically lose a small portion of fees. Over time and across many withdrawals, this leads to measurable revenue leakage for the protocol treasury.

Recommendation: Use `Math.Rounding.Ceil` in the fee calculation to ensure that the protocol always collects the full intended fee:

[L-08] Withdraw from staking limit can be bypassed via multiple calls

The function `withdrawFromStaking()` checks that the `amount` does not exceed `hypeRequestedForWithdraw()` to make sure only the required amount of assets are withdrawn from staking vault to `WithdrawManager`. However, this limit is only enforced per individual call and not across the total sum of multiple withdrawals. As a result, a protocol admin can repeatedly call `withdrawFromStaking()` with amounts just below the limit, eventually withdrawing more than the total `hypeRequestedForWithdraw`.

```solidity function withdrawFromStaking(uint256 amount) external { if (! roleRegistry.hasRole(roleRegistry.PROTOCOL_ADMIN(), msg.sender)) revert NotAuthorized(); if (amount > IWithdrawManager(withdrawManager).hypeRequestedForWithdraw()) revert ExceedsLimit();

```
if (block.timestamp < lastWithdrawalTimestamp + withdrawalCooldownPeriod) {
    revert WithdrawalCooldownNotMet();
}

_encodeAction(5, abi.encode(_convertTo8Decimals(amount)));
lastWithdrawalTimestamp = block.timestamp;
emit HyperCoreStakingWithdraw(amount);
```

} ````

This allows the total withdrawn amount to exceed the actual requested withdrawals, potentially draining more funds from staking than should be available for user redemptions.

[L-09] Funds not transferred before `withdrawManager` update

The `setWithdrawManager()` function can be used by the protocol upgrader to change the withdraw manager address:

```
    function setWithdrawManager(address _withdrawManager) external {
        roleRegistry.onlyProtocolUpgrader(msg.sender);
        withdrawManager = _withdrawManager;

        emit WithdrawManagerUpdated(withdrawManager);
    }
```

However, this function fails to send the accumulated `hypeRequestedForWithdraw` back to the current staking contract before replacing it.

Hence, it is recommended to clear the requested funds for withdrawal before changing. This can be achieved by transferring the requested funds to the staking core and clearing `hypeRequestedForWithdraw` or by directly transferring the requested amount to the new withdrawal manager contract.

[L-10] `depositToHyperCore()` fails to account for `hypeRequestedForWithdraw`

The `depositToHyperCore()` function allows the protocol admin to deposit HYPE tokens into HyperCore:

```
    function depositToHyperCore(uint256 amount) external {
        if (!roleRegistry.hasRole(roleRegistry.PROTOCOL_ADMIN(), msg.sender)) revert
NotAuthorized();
        uint256 truncatedAmount = amount / 1e10 * 1e10;
        if (amount != truncatedAmount) {
            revert PrecisionLossDetected(amount, truncatedAmount);
        }

        (bool success,) = payable(L1_HYPE_CONTRACT).call{value: amount}("");
        if (!success) revert FailedToDepositToHyperCore();

        lastHyperCoreOperationBlock = block.number;
        emit HyperCoreDeposit(amount);
    }
```

However, this function fails to account for `hypeRequestedForWithdraw`, which signifies the amount of HYPE that should remain untouched for finalising withdrawals. Hence, unintentionally, `finalizeWithdrawals()` might fail even when they are ready to be finalised, in a case where the `StakingCore` contract lacks sufficient funds.

A check can be observed at `withdrawFromStaking()` to disallow excess withdrawals from staking:

```
    function withdrawFromStaking(uint256 amount) external {
        if (!roleRegistry.hasRole(roleRegistry.PROTOCOL_ADMIN(), msg.sender)) revert
NotAuthorized();
        if (amount > IWithdrawManager(withdrawManager).hypeRequestedForWithdraw()) revert
ExceedsLimit();       <<@
```

It is recommended to add a similar check for `depositToHyperCore()` as well.

[L-11] First staker uses wrong `exchangeRatio` if `totalSupply()` of beHYPE is 0

`updateExchangeRatio()` is used to calculate the exchange ratio between beHYPE and HYPE, and it is called by the admin to update it.

```
function updateExchangeRatio() external {
        if (!roleRegistry.hasRole(roleRegistry.PROTOCOL_ADMIN(), msg.sender)) revert
NotAuthorized();

        uint256 blocksPassed = block.number - lastHyperCoreOperationBlock;
        if (blocksPassed < MIN_BLOCKS_BEFORE_EXCHANGE_RATIO_UPDATE) {
            revert ExchangeRatioUpdateTooSoon(MIN_BLOCKS_BEFORE_EXCHANGE_RATIO_UPDATE,
blocksPassed);
        }

        uint256 totalProtocolHype = getTotalProtocolHype();

@>      uint256 newRatio = Math.mulDiv(totalProtocolHype, 1e18, beHypeToken.totalSupply());

        uint256 ratioChange;
        if (newRatio > exchangeRatio) {
            ratioChange = newRatio - exchangeRatio;
        } else {
            ratioChange = exchangeRatio - newRatio;
        }

        uint256 elapsedTime = block.timestamp - lastExchangeRatioUpdate;

        if (elapsedTime == 0) revert ElapsedTimeCannotBeZero();

        uint256 percentageChange = Math.mulDiv(ratioChange, 1e18, exchangeRatio);
        uint256 yearlyRate = Math.mulDiv(percentageChange, 365 days, elapsedTime);
        uint256 yearlyRateInBps = yearlyRate / 1e14;

        uint16 yearlyRateInBps16 = uint16(Math.min(yearlyRateInBps, type(uint16).max));

        if (exchangeRateGuard) {
            if (yearlyRateInBps16 > acceptablAprInBps) revert
ExchangeRatioChangeExceedsThreshold(yearlyRateInBps16);
        }

        uint256 oldRatio = exchangeRatio;
        exchangeRatio = newRatio;
        lastExchangeRatioUpdate = block.timestamp;

        emit ExchangeRatioUpdated(oldRatio, exchangeRatio, yearlyRateInBps16);
    }
```

As you can see, the rate is calculated by dividing the total HYPE tokens by the total supply of beHYPE. The problem arises if the total supply of beHYPE is 0 (meaning all tokens have been burned). In this case, when the admin calls `updateExchangeRatio()` to update the ratio, the transaction will always revert due to a division by zero.

This prevents the exchange ratio from being updated, causing the first new staker to rely on the outdated ratio. As a result, they may mint an incorrect amount of beHYPE, either more or less than intended.

Let's illustrate the issue with an example. Assume the contract holds `100e18` HYPE tokens and `50e18` beHYPE tokens, so the `exchangeRatio = 100e18 / 50e18 = 2`.

1. Bob is the last user holding `50e18` beHYPE.
2. Bob calls `withdraw()` and burns `50e18` beHYPE, receiving `100e18` HYPE.
3. The admin then calls `updateExchangeRatio()` to update the ratio, but it reverts because the total supply of beHYPE is now 0.
4. Alice calls `stake()` with `100e18` HYPE, but since the ratio wasn't updated, the old value (2) is used.
5. Alice receives only `50e18` beHYPE (`100e18 / 2`) instead of `100e18` beHYPE (initial ratio when beHYPE supply is 0 is 1:1).

As a result, Alice gets only half the beHYPE she should, because the ratio couldn't be updated.

The issue can also be reproduced with the following POC. To run it, copy the code into `WithdrawManager.t.sol`.

```
function test_CantUpdateExchangeRatio() public {

        uint256 oldExchangeRatio = stakingCore.exchangeRatio();

        vm.startPrank(user);
        beHYPE.approve(address(withdrawManager), 1 ether);
        withdrawManager.withdraw(1 ether, false, 0.9 ether);
        vm.stopPrank();

        vm.startPrank(user2);
        beHYPE.approve(address(withdrawManager), 10 ether);
        withdrawManager.withdraw(10 ether, false, 9 ether);
        vm.stopPrank();

        vm.prank(admin);
        withdrawManager.finalizeWithdrawals(2);

        vm.warp(block.timestamp + (365 days * 100));
        vm.roll(block.number + 5);

        vm.expectRevert("panic: division or modulo by zero (0x12)");
        vm.prank(admin);
        stakingCore.updateExchangeRatio();

        uint256 actualExchangeRatio = stakingCore.exchangeRatio();

        assertEq(oldExchangeRatio, actualExchangeRatio);
    }
```

Recommendations

If the `totalSupply()` of beHYPE is 0, return the initial `exchangeRatio` of `1e18` (1:1 ratio).

[L-12] Loss events may be blocked from updating exchange ratio due to APR guard

The `updateExchangeRatio()` function uses `acceptablAprInBps` as a guardrail for limiting sudden changes in the exchange ratio. The same threshold is applied whether the change reflects profits or losses:

```solidity if (exchangeRateGuard) { if (yearlyRateInBps16 > acceptablAprInBps) revert ExchangeRatioChangeExceedsThreshold(yearlyRateInBps16); } ````

This design works for **profits**, which accumulate gradually over time, but fails for **losses**, which can happen suddenly (e.g., a slashing event in HyperCore).

If a slash causes a sharp drop in protocol value, the resulting ratio change may easily exceed the acceptable APR threshold. Instead of updating the exchange ratio to reflect the loss, the function reverts. This prevents the ratio from ever being updated to the correct (lower) value, leaving the system in a stale, inflated state.

Recommendations

Use **separate thresholds** for positive and negative changes. Losses should never be blocked from being reflected in the exchange ratio.

[L-13] Rounding in `stake()` is in favour of users

The staking flow mints BeHYPE for deposited HYPE based on the current `exchangeRatio`. The final rounding logic applied is in favor of user: In the `stake()` function, BeHYPE is minted as follows:

```
function stake(string memory communityCode) public payable {
    if (paused()) revert StakingPaused();

    beHypeToken.mint(msg.sender, HYPEToBeHYPE(msg.value));
    emit Deposit(msg.sender, msg.value, communityCode);
}
````

The conversion from HYPE to BeHYPE uses:
```solidity
function HYPEToBeHYPE(uint256 HYPEAmount) public view returns (uint256) {
    return Math.mulDiv(HYPEAmount, 1e18, exchangeRatio);
}
```

Here, the `exchangrRatio` is the denominator. Because it has already been rounded down, dividing by a smaller denominator inflates the result, granting the user slightly more BeHYPE than fair value.
```

This rounding discrepancy compounds over time: every stake mints marginally more BeHYPE than it should.

Note: exchange ratio is rounded down when being update:

```
    function updateExchangeRatio() external {
        ...
        uint256 newRatio = Math.mulDiv(totalProtocolHype, 1e18, beHypeToken.totalSupply());
        ...
```

Recommendations

- in `HYPEToBeHYPE` function use rounded up `exchangeRatio`.
- in `BeHYPEToHYPE` function use rounded down `exchangeRatio`.

[L-14] Pending withdrawals are not considered in total Hype assets

In the current withdrawal flow, when a user requests a non-instant withdrawal, their BeHYPE tokens are transferred to the `WithdrawManager` but are not burned until the withdrawal is finalized.

```
// withdraw() - non-instant branch
withdrawalQueue.push(WithdrawalEntry({
    user: msg.sender,
    beHypeAmount: beHypeAmount,
    hypeAmount: hypeAmount,
    claimed: false
}));
userWithdrawals[msg.sender].push(withdrawalId);

// Tokens are transferred but not yet burned
beHypeToken.transferFrom(msg.sender, address(this), beHypeAmount);
````

Burning only happens later in `finalizeWithdrawals()`:

```solidity
// finalizeWithdrawals()
beHypeToken.burn(address(this), beHypeAmountToFinalize);
stakingCore.sendFromWithdrawManager(hypeAmountToFinalize, address(this));
```

However, the tokens being withdrawn are not excluded from `StakingCore` vault accounting and remain part of total Hype assets and total BeHype supply, until the order is finalized. This means that while the user's redemption value is fixed at the time of withdrawal request (`hypeAmount`), the tokens continue to accrue yield or loss within the vault.

When the user later finalizes the order, only the originally recorded asset amount is transferred, while the yield that accumulated during the pending period remains in the vault. This causes the withdrawing user to lose a portion of their profits, and those profits are instead redistributed to the remaining participants all in once.

This sudden raise in total Hyper assets of vault can be exploited by malicious users.

Recommendations

Update the withdrawal logic so that pending withdrawals are excluded from vault accounting immediately. The simplest solution is to burn `BeHYPE` tokens at withdrawal request time, thereby fixing the user's claim and ensuring they do not continue accruing rewards or losses after initiating a withdrawal.

[L-15] `StakingCore` does not account for pending deposits to Hyper Core

The function `getTotalProtocolHype()` fails to include deposits made to Hyper Core within the **current block**.

When users call `StakingCore.depositToHyperCore`, tokens are transferred to the bridge and remain in a **pre-credit state** until the next block. During this period, `getTotalProtocolHype()` under-reports the protocol balance because it only reflects credited balances.

```
    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);

        uint256 totalHypeInLiquidityPool = address(this).balance;

        uint256 total = totalHypeInStakingAccount + totalHypeInSpotAccount +
totalHypeInLiquidityPool;
        return total;
    }
```

As a result, the protocol state is inconsistent: in the block of deposit, tokens are already locked in the bridge but not reflected in `getTotalProtocolHype()`. This discrepancy may lead to inaccurate protocol calculations (e.g., staking rewards, governance weight, or logic relying on total hype supply).

Recommendations

Track deposits to Hyper Core made in the current block and add them to the `getTotalProtocolHype` immediately.
Then, remove these tracked pending deposits once they are credited in the next block to prevent double-counting.