HORD

SMART CONTRACT AUDIT

zokyo

September 19th 2023 | v. 2.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.

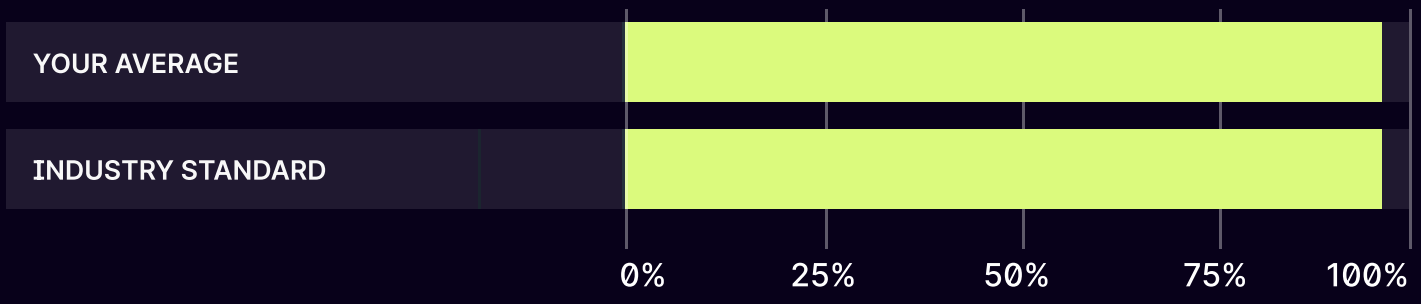
TECHNICAL SUMMARY

This document outlines the overall security of the Hord smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Hord smart contract codebase for quality, security, and correctness.

Contract Status

Testable Code



Correspond industry standards.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Hord team put in place a bug bounty program to encourage further active analysis of the smart contract.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Hord repository:
https://github.com/hord/smart-contracts

Branch: develop
Initial commit: 61281ed4857245f562fd35ca9b421be9c4e47005
Final commit: efb71eb817291972822e05a66251a795c3c43663

Check for changes, re-audit after 1 iteration of audit, use first version code to check contract upgrade during testing:

Previous/First version of code commit: c3b0e02e6b42f2ffe524f3f36990d459e0555533

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- StakingConfiguration.sol (87 lines changed)
- HordETHStakingManager.sol (377 lines changed)*
- HordETHStakingWithdrawalManager.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Hord smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contract logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contracts by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

Zokyo Security received the contracts, a part of the Hord project, for a security audit. The protocol represents a staking and withdrawal system for Hord Ethereum (HETH) tokens, including fee calculations and debt handling implementation. In this iteration of the audit, exactly the withdrawal part was audited.

The audit's goal was to verify that the contract functions correctly with a known level of security and to check the code line by line. Our auditors also checked the contract's code against their own checklist of vulnerabilities, validated the business logic of the contract, and ensured that best practices in terms of gas spending were applied. Several medium-severity and low-severity issues were found during the manual part of the audit. One of the medium-severity issues was connected to inefficient looping, while the other one was related to an unrestricted protocol fee. Both of these issues were successfully fixed. However, there were also unresolved issues, such as a lack of validation and inefficient removal.

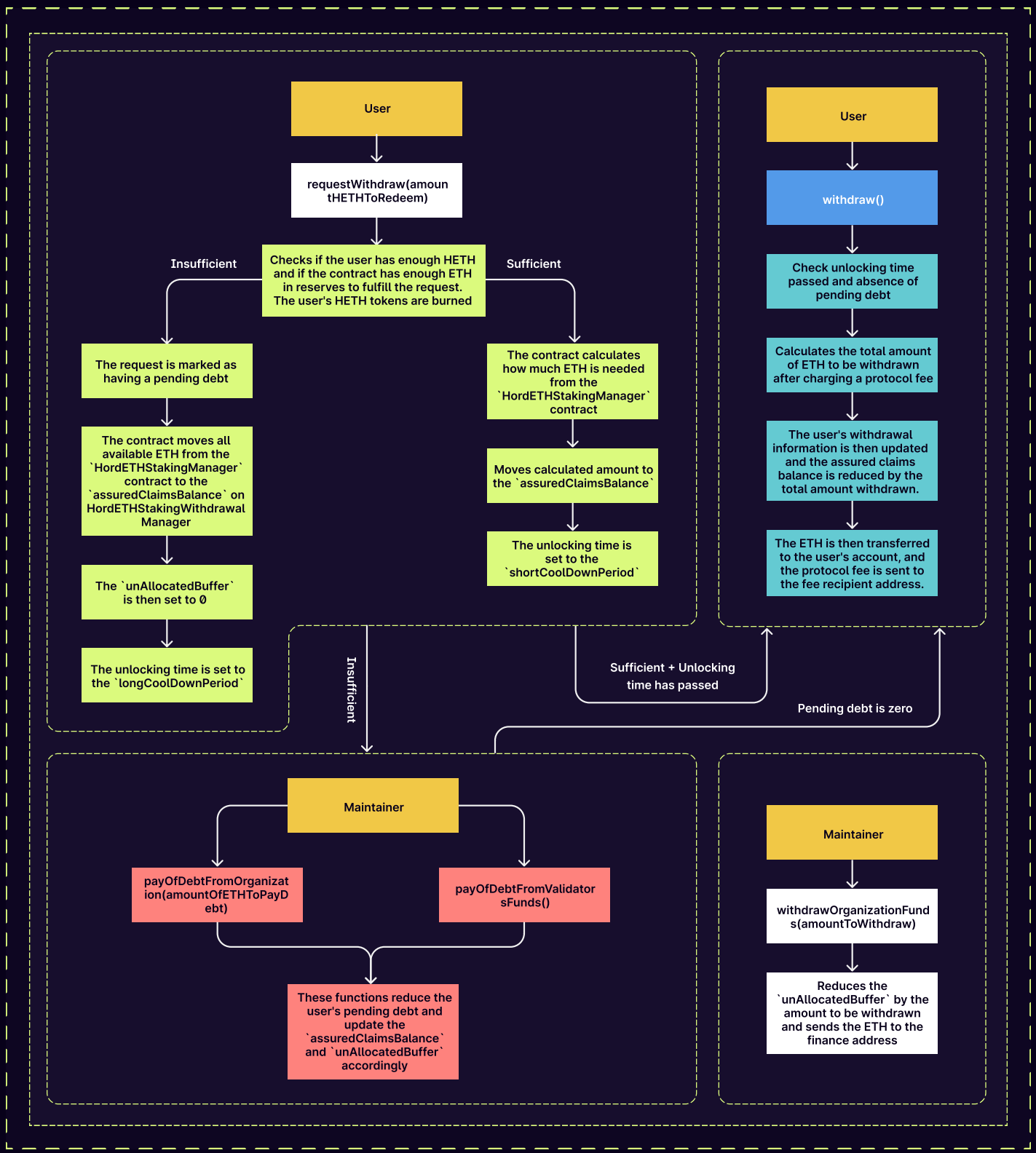
Another part of the audit process was to check the native tests prepared by the Hord team and prepare a set of custom test scenarios. It should be mentioned that the Hord team had provided a solid set of tests covering all the contract's core logic. The complete set of unit tests can be seen in the Code Coverage and Test Result sections.

The total security of the contract is high enough. The contract is well-written and tested. The basic staking and withdrawal functionality and additional logic written by the Hord team were carefully audited.

As an important note, while the smart contract part of the Hord project underwent a thorough audit, the backend part was not audited. The backend part is also critical as user funds depend on it. Therefore, while this audit provides a high level of assurance about the security and functionality of the smart contracts, it does not cover the entire system. We recommend a thorough audit of the backend to ensure the overall security of the Hord project.

In addition to the security audit, the migration or upgrade process from one contract version to another has also been thoroughly tested. This verifies that the upgrade process does not introduce new vulnerabilities or issues.

In conclusion, the Hord contracts function correctly with a known level of security. The contracts work as expected, and the tests cover all the core logic, ensuring the contract's functionality and security. Despite a few areas for improvement, the contract is well-written and secure. The protocol has good documentation, which can be viewed at this link: https://docs.hord.fi/


User
requestWithdraw(amountHETHToRedeem)
Checks if the user has enough HETH and if the contract has enough ETH in reserves to fulfill the request. The user's HETH tokens are burned
Insufficient
Sufficient
The request is marked as having a pending debt
The contract moves all available ETH from the `HordETHStakingManager` contract to the `assuredClaimsBalance` on HordETHStakingWithdrawalManager
The `unAllocatedBuffer` is then set to 0
The unlocking time is set to the `longCoolDownPeriod`
The contract calculates how much ETH is needed from the `HordETHStakingManager` contract
Moves calculated amount to the `assuredClaimsBalance`
The unlocking time is set to the `shortCoolDownPeriod`
User
withdraw()
Check unlocking time passed and absence of pending debt
Calculates the total amount of ETH to be withdrawn after charging a protocol fee
The user's withdrawal information is then updated and the assured claims balance is reduced by the total amount withdrawn.
The ETH is then transferred to the user's account, and the protocol fee is sent to the fee recipient address.
Insufficient
Sufficient + Unlocking time has passed
Pending debt is zero
Maintainer
payOfDebtFromOrganization(amountOfETHToPayDebt)
payOfDebtFromValidatorsFunds()
These functions reduce the user's pending debt and update the `assuredClaimsBalance` and `unAllocatedBuffer` accordingly
Maintainer
withdrawOrganizationFunds(amountToWithdraw)
Reduces the `unAllocatedBuffer` by the amount to be withdrawn and sends the ETH to the finance address

Description

The `HordETHStakingWithdrawalManager` contract is part of the Hord project, which is a decentralized platform for social trading. This contract manages the withdrawal of HordETH (HETH) tokens staked by users. Users can request to withdraw their staked tokens, which are then processed by the contract. The contract also handles the conversion of HETH to ETH and manages the buffer of unallocated ETH.

This contract also interacts with others such as `HordETHStakingManager`, `StakingConfiguration`, and `HordCongress` to manage the staking and withdrawal process. It also emits several events to track the state of withdrawals and deposits.

Withdrawal flow:

1. Request Withdrawal: A user initiates a withdrawal request by calling the `requestWithdraw` function. The user specifies the amount of HETH they wish to redeem. The function checks if the user has enough HETH and if the contract has enough ETH in reserve to fulfill the request. The user's HETH tokens are burned. If the reserves are insufficient, the request is marked as having a pending debt.

2. Update Reserves: If the reserves are insufficient to meet the withdrawal request, the contract transfers all available ETH from the HordETHStakingManager contract to the assuredClaimsBalance, setting the unAllocatedBuffer to 0. However, if the reserves are sufficient, the contract determines the necessary amount of ETH from the HordETHStakingManager contract and transfers this amount to the assuredClaimsBalance.

3. Unlocking Time: The contract establishes a withdrawal unlocking time based on whether the contract has sufficient reserves to meet the withdrawal request. If the reserves are sufficient, the unlocking time is set to the shortCoolDownPeriod. Conversely, if the reserves are insufficient, the unlocking time is set to the longCoolDownPeriod.

4. Finalize Withdrawal: Once the unlocking time has elapsed, the user can invoke the withdraw function to complete the withdrawal. This function verifies if the unlocking time has passed and if the user has no outstanding debt. It then computes the total amount of ETH to be withdrawn, considering a protocol fee. The user's withdrawal information is updated accordingly, and the total withdrawal amount decreases the assured claims balance. The ETH is subsequently transferred to the user's account, with the protocol fee being sent to the designated fee recipient address.

5. Pay Off Debt: If the user has a pending debt, the contract can call the `payOfDebtFromOrganization` or `payOfDebtFromValidatorsFunds` functions to settle the debt. These functions decrease the user's pending debt and adjust the assuredClaimsBalance and unAllocatedBuffer accordingly.

6. Withdraw Organization Funds: The contract also allows for the withdrawal of organization funds by calling the `withdrawOrganizationFunds` function. This function reduces the unAllocatedBuffer by the amount to be withdrawn and sends the ETH to the finance address.

Withdrawal flow:

1. Request Withdrawal: A user can initiate a withdrawal request by invoking the requestWithdraw function, specifying the amount of HETH they wish to redeem. The function verifies if the user possesses sufficient HETH and if the contract has adequate ETH reserves to meet the request. The user's HETH tokens are subsequently burned. If the reserves are insufficient, the request is marked as having a pending debt.

2. Update Reserves: If the reserves cannot fulfill the withdrawal request, the contract moves all available ETH from the HordETHStakingManager contract to the assuredClaimsBalance. The unAllocatedBuffer is then set to `0`. If the reserves are sufficient, the contract calculates how much ETH is needed from the HordETHStakingManager contract and moves this amount to the assuredClaimsBalance.

3. Unlocking Time: Depending on whether the contract has adequate reserves to satisfy the withdrawal request, it sets an unlocking time for the withdrawal. If the reserves are sufficient, the shortCoolDownPeriod is set as the unlocking time. However, if the reserves are insufficient, the unlocking time is set to the longCoolDownPeriod.

4. Finalize Withdrawal: After the unlocking time has passed, the user can call the `withdraw` function to finalize the withdrawal. This function verifies if the unlocking period has passed and if the user has no pending debt. It then calculates the total amount of ETH to be withdrawn after charging a protocol fee. The user's withdrawal information is updated, and the assured claims balance is decreased by the total withdrawal amount. The ETH is then transferred to the user's account, and the protocol fee is sent to the fee recipient address.

5. Pay Off Debt: If the user has a pending debt, the contract can call the `payOfDebtFromOrganization` or `payOfDebtFromValidatorsFunds` functions to settle the debt. These functions decrease the user's pending debt and adjust the assuredClaimsBalance and unAllocatedBuffer accordingly.

6. Withdraw Organization Funds: The contract also allows for the withdrawal of organization funds through the `withdrawOrganizationFunds` function. This function reduces the unAllocatedBuffer by the amount to be withdrawn and sends the ETH to the finance address.

Deployment

The deployment scripts compile the contracts, deploy them to the specified network, and save the contract addresses for later use.

The first script deploys `StakingConfiguration` and `HordETHStakingManager` contracts. Subsequently, a second script deploys `HordETHStakingWithdrawalManager` contract as a proxy, utilizing the Upgrades plugin from OpenZeppelin.This approach is considered a good practice, enabling future contract upgrades without state loss.

However, it's essential to ensure that the saveContractAddress and saveContractProxies functions are correctly implemented to store the contract addresses and proxies. These functions are vital in maintaining a record of your deployed contracts. It's crucial to rigorously test deployment scripts on a test network before proceeding with deployment on the mainnet.

Here are some suggestions to improve the deployment scripts:

1.Set the new settings: After the contract upgrade, it's necessary to establish the new settings for the contract. This includes the addresses of `StakingConfiguration`, `HordETHStakingManager`, `MaintainersRegistry`, and `HordCongress` contracts, the protocol fee, the cool-down periods, and the finance address.

2.Verify the new settings: After setting the new settings, verify that they have been correctly set. You can add checks in your deployment scripts to ensure this.

3.Upgrade the proxy: Use the `upgradeProxy` function from OpenZeppelin's Upgrades plugin to upgrade the contract. This function deploys a new implementation contract, updates the proxy to point to this unique implementation, and preserves the state from the old contract. Notably, the `upgradeProxy` function also verifies if the contracts are upgrade-compatible and ensures that the storage remains intact. Therefore, it's recommended to use `upgradeProxy` instead of other methods for upgrading contracts. However, prior to upgrading, confirm that the contracts are verified on the mainnet; otherwise, an error may occur. Alternatively, you can use the `forceImport` function with the implementation as a parameter located on the mainnet, ensuring that the latest implementation is used before upgrading.

This way, you don't have to bind the implementation to the proxy manually, and you don't have to re-verify the contract on Etherscan.
As for now, the implementations are deployed in the upgradeStakingConfiurationAndManager.js. However, they are not set in the corresponding proxy contracts.

4.Numbering the deployment scripts can be helpful for better interaction. It allows you to keep track of the deployment order and makes it easier to automate the deployment process.

Settings

1. **StakingConfiguration Contract Address**: This is the address of the `StakingConfiguration` contract which is set during the initialization of the `HordETHStakingWithdrawalManager` contract. This contract provides important parameters like `shortCoolDownPeriod`, `longCoolDownPeriod`, `withdrawalProtocolFee`, and `feeRecipient`. If this is not set correctly, the withdrawal process will not function as expected.

2. **HordETHStakingManager Contract Address**: This is the address of the `HordETHStakingManager` contract which is also set during the initialization of the `HordETHStakingWithdrawalManager` contract. This contract is responsible for managing the staking and unstaking of HETH tokens. If this is not set correctly, users will be unable to stake or unstake their HETH tokens.

3. **MaintainersRegistry Contract Address**: This is the address of the `MaintainersRegistry` contract which is set during the initialization of the `HordETHStakingWithdrawalManager` contract. This contract checks if a function call is initiated by a maintainer. If this is not set correctly, unauthorized addresses will be able to call functions restricted to maintainers.

4. **HordCongress Contract Address**: This is the address of the `HordCongress` contract which is set during the initialization of the `HordETHStakingWithdrawalManager` contract. This contract checks if a function call is initiated by the HordCongress. If this is not set correctly, unauthorized addresses will be able to call functions restricted to the HordCongress.

5. **Protocol Fee**: This is the fee charged by the protocol for withdrawals. It is set in the `StakingConfiguration` contract and deducted from the total amount of ETH to be withdrawn by the user. If this is set too high, it may discourage users from withdrawing their funds.

6. **Cool Down Periods**: These are the periods a user has to wait before they can withdraw their funds after making a withdrawal request. They are set in the `StakingConfiguration` contract. If these are set too long, it may discourage users from staking their HETH tokens.

7. **Finance Address**: This is the address to which the protocol fee for withdrawals is sent. It is set in the `StakingConfiguration` contract. If this is not set correctly, the protocol fees will be sent to an incorrect address.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

MEDIUM-1 | RESOLVED

Inefficient Looping.

HordETHStakingWithdrawalManager.sol: payOfDebtFromOrganization(), payOfDebtFromValidatorsFunds().
The functions employ a for loop to traverse the withdrawalDebts storage array. If the array grows excessively large, it could result in high gas costs and potential out-of-gas errors. Although the break instruction is executed when the value of amountOfETHToPayDebt or msg.value decreases, this may not always be reliable, particularly if functions are invoked from a backend that doesn't estimate possible reverts due to gas issues.

Recommendation:
Implement a mechanism where the maintainer can specify a range of elements to process in each transaction. This would allow the function to handle large arrays more efficiently by processing them in smaller chunks.

Post-audit:
Mechanism was implemented.

Unrestricted protocol fee.

HordETHStakingWithdrawalManager.sol: withdraw() Function: The protocol fee for withdrawal is set at a fixed rate and is not calculated based on the withdrawal amount. This could lead to scenarios where the fee surpasses the amount a user intends to withdraw. In such instances, the protocol would need to reduce the fee to enable the user's withdrawal, which could potentially be exploited by other users.

Recommendation:

Implement a dynamic fee structure where the fee is calculated as a percentage of the withdrawal amount. This approach would prevent the fee from exceeding the withdrawal amount and eliminate the need for manual fee adjustments. Alternatively, within the requestWithdraw() function, verify if the withdrawal amount the user intends to withdraw is greater than or equal to the protocol fee. However, this option might be less preferable than a dynamic fee as it could restrict users with a small amount of HETH from redeeming it.

Post-audit:

A dynamic fee system was implemented in which two percentages are involved, one for transactions with a small amount and one with a large amount. An amount limit was also added, after which a transaction will be considered as a transaction with a high amount.

LOW-1 | VERIFIED

Lack of a function to update the addresses of associated contracts.

HordETHStakingManager.sol, HordETHStakingWithdrawalManager.sol
The contracts do not have a function to update the staking configuration contract. If the staking configuration contract needs to be updated or replaced, it would not be possible with the current contract.
Also, HordETHStakingWithdrawalManager.sol doesn't have a function for hordETHStakingManager updating.

Recommendation:
Implement a function to update the associated contracts.

From client:
If a situation arises where a new contract must be deployed and a new address needs to be set, The existing contracts will be upgraded and methods for setting new addresses will be added.

Lack of validation.

HordETHStakingWithdrawalManager.sol: requestWithdraw(), withdrawOrganizationFunds();
StakingConfiguration.sol: initialize(), setMaxCap(), setGenerationRate(), setAmountNeededToLaunchNewValidator(), setAmountETHInValidator(), setPercentFeeFromWithdrawal(), setPercentFeeFromWithdrawalMicroTx(), setEthWithdrawalMicroTxThreshold();
HordETHStakingManager.sol: launchNewValidatorWithSSV(), userDepositETH().
Either not all or none of the parameters passed to these functions are validated. This may lead to unexpected behavior in further use, which is why it is recommended to validate parameters. For example, addresses should not be equal to zero addresses, and numeric parameters should be validated to ensure they are not equal to `0`.

Recommendation:
Add the necessary validation.

From client:
There is validation for parameters for both mentioned functions from the HordETHStakingWithdrawalManager contract. There is no need to add validations for the mentioned functions from the StakingConfiguration contract because some of them are not currently used, and for some, we don't even have any restrictions; it can be `0` or some large number. For the launchNewValidatorWithSSV() function, all parameters are checked on the SSV and beaconchain deposit contracts, so there is no need for a double check.

Post-audit:
In the HordETHStakingWithdrawalManager function withdrawOrganizationFunds, to have only external validation by admin is normal practice, but we recommend adding validation inside the contract for functions that users will call. In the requestWithdraw function, there is no validation for the user to pass `0` amount. This was also verified in the tests. Though the issue was marked as verified, we still recommend the Hord team add necessary validations in the code of smart contracts.

INFORMATIONAL-2 | RESOLVED

Solidity style guide violation.

StakingConfiguration.sol.
Smart contract contains several magic numbers, which are hardcoded values that may not be easily understood or changed. These magic numbers can make the code harder to read, maintain, and modify.

Recommendation:
Replace magic numbers with named constants or variables to improve code readability and maintainability.

Post-audit:
Magic numbers were replaced with variables.

INFORMATIONAL-3 | RESOLVED

To-dos and commented out code in the project codebase.

ordETHStakingManager.sol: line 360, 379
The to-dos in the project code signal that the project contains unfinished functionality and unimplemented flows/edge cases. Thus, that may lead to unstable behavior, mismatch with the documentation, incorrect business logic, and missed edge cases.

Recommendation:
Resolve the to-dos in code.

Post-audit.:
To-dos were resolved.

INFORMATIONAL-4 | VERIFIED

Inefficient removal.

HordETHStakingWithdrawalManager.sol: removeUserFromWithdrawalDebts().
The function removeUserFromWithdrawalDebts() is implemented in a way that it shifts all elements of the array after the removed element. This operation has a linear time complexity and can be inefficient when the array grows large.

Recommendation:
Swap the element to be removed with the last element in the array, then call pop() on the array. This would remove the element in constant time. Otherwise, verify if the order of users in the array is important.

From client:
Order of users in array is important, therefore, mechanism with a shift will not be changed.

INFORMATIONAL-5 | RESOLVED

Inconsistency in zero handling.

HordETHStakingManager.sol: getAmountOfETHForHETH().
In the getAmountOfHETHforETH function, there is a condition that checks if the total supply is zero, and if so, it returns the input amount of ETH. However, in the getAmountOfETHForHETH function, there is no such condition to handle a zero total supply. This inconsistency could lead to unexpected behavior or errors if the total supply is zero when the functions are called.

Recommendation:
Add a condition to handle a zero total supply in the getAmountOfETHForHETH function, similar to what is done in the getAmountOfHETHforETH function. This would ensure consistent behavior between the two functions and prevent potential division by zero errors.

Post-audit:
Condition was added.

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Hord in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the code's functionality and a review of the Hord contract requirements for details about issuance amounts and how the system handles these.

```
HordETHStakingWithdrawalManager
    # Withdrawal
        with two depositors
    ✓ Withdraws by the first depositor (276ms)
    ✓ Withdraws by the second depositor with closing out the debt to users because of a closed validator
WithdrawalUpgradeFork
    # Withdrawal
        with two depositors
    ✓ Withdraws by the first depositor (748ms)
WithdrawalScenarios
    # Withdrawal
        with two depositors
    ✓ Withdraws with protocol fee
    ✓ Withdraws and verification of shares calculation
    # Withdrawal
        with four+ depositors
    ✓ Stakes and withdraws with a lot of users
    ✓ Withdraws with a lot of users (amount ETH to withdraw > total ETH reserves)
    ✓ Request 0 amount for withdrawal
```

Note.
We have thoroughly examined the withdrawal functionality and its interaction with staking in the HordETHStakingWithdrawalManager smart contract. The comprehensive tests covered a variety of withdrawal scenarios, ensuring that the contract behaves as expected under different conditions.
We tested the staking and withdrawal process with a large number of users. This helped us confirm that the contract could manage high volumes of staking and withdrawal transactions efficiently.

	HordETHStakingWithdrawalManager.sol HordETHStakingManager.sol StakingConfiguration.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Hord team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Hord team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo