Security Review Report
NM-0225 Summon

NETHERMIND
SECURITY

(Apr 16, 2024)

Contents

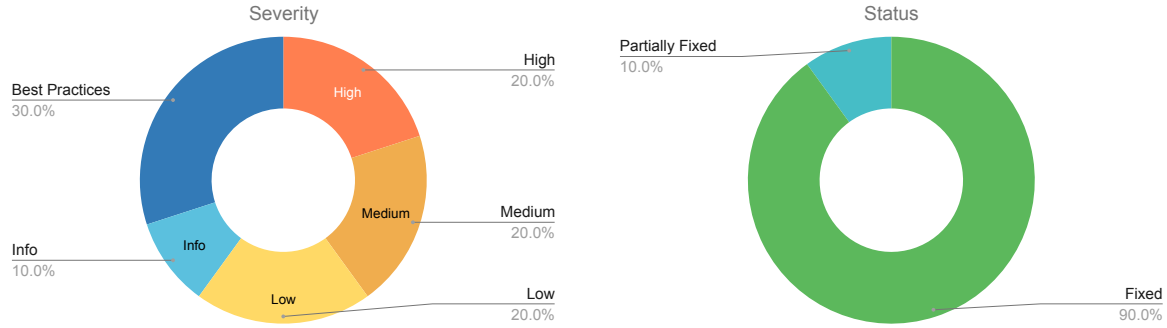
1 Executive Summary

This document outlines the security review conducted by Nethermind Security for Summon. The **Summon** system allows gamifying the participation and commitment of users towards a community, using playful reward mechanisms in the form of tokens to obtain the engagement of its members. The system supports various token standards for representing user rewards, including ERC-20, ERC-721, and ERC-1155, and it can also utilize ETH. This code review focuses on the `LootDrop.sol` contract, which contains the logic for the main system functions related to managing rewards for users utilizing ERC-1155 tokens.

> **Note about the audit**
>
> It is important to mention that the audit was conducted within a tight time constraint of 3 days to meet a launch plan. Therefore, the protocol should consider an after-deployment audit for additional audit coverage of the codebase.

The audited code comprises 409 lines of code in Solidity. The Summon's team provided a detailed description of the functionality of **LootDrop** contract, with supporting diagrams of interaction flows. The audit was performed using: (a) manual analysis of the codebase, (b) simulation of the smart contracts. **Along this document, we report** 10 points of attention, where two are classified as `High`, two are classified as `Medium`, two are classified as `Low`, and four 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 of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. 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) distribution of issues according to the severity

(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (0), **High** (2), **Medium** (2), **Low** (2), **Undetermined** (0), **Informational** (1), **Best Practices** (3). **(b) Distribution of status: Fixed** (9), **Partially Fixed** (1), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Apr 12, 2024
Final Report	Apr 16, 2024
Methods	Manual Review
Repository	achievo-contracts
Commit Hash	44fcac04069938ae1177560c40bafff6fd61f7e3
Final Commit Hash	5ab2e613977a6bf25179e5f001563268b25b645d
Documentation	Comments in the codebase and diagrams
Documentation Assessment	Low
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	LootDrop.sol	409	39	9.5%	79	527
	Total	**409**	**39**	**9.5%**	**79**	**527**

3 Summary of Issues

	Finding	Severity	Update
1	Distribution of rewards could revert for ERC721	High	Fixed
2	Possible signature replay attack across different chains	High	Fixed
3	Lack of ERC20 transfer safety checks in LootDrop contract	Medium	Fixed
4	User could claim rewards even if contract is paused	Medium	Fixed
5	Centralization risks	Low	Partially Fixed
6	`itemIds` can be duplicated in `createTokenAndDepositRewards(...)`	Low	Fixed
7	ERC1155 with token ID zero can't be added as a reward	Info	Fixed
8	Missing input validations	Best Practices	Fixed
9	Perform input parameter validation before modifying the contract state	Best Practices	Fixed
10	Unused Code	Best Practices	Fixed

4 System Overview

The **Summon** protocol introduces the **LootDrop** contract, enabling users to earn rewards in various forms, such as ERC20 tokens, ERC721, ERC1155 items, and Ether, based on predefined allocations. The Manager role-based account is authorized to create these token rewards, and users with reward tokens can claim them by invoking the mint function with a pre-approved whitelist signer signature. Rewards can also be airdropped to users through Minter and Manager role-based accounts.

The system distinguishes between 5 main roles:

Manager Role: It is an account with elevated privileges that can perform the below administrative tasks on the protocol:

- createTokenAndDepositRewards(...)
- createMultipleTokensAndDepositRewards(...)
- withdrawAssets(...)
- updateClaimRewardPaused(...)
- updateTokenMintPaused(...)
- pause(...)
- unpause(...)
- adminClaimReward(...)

Minter Role: It is an account with elevated privileges that can perform the below administrative tasks on the protocol:

- adminBatchMintById(...)
- adminMintById(...)
- adminMint(...)

Whitelist Signer: The whitelist signer account is authorized to sign the whitelist signature so that users can claim reward tokens.

Dev Config Role: It is an account with elevated privileges that can perform the below administrative tasks on the protocol:

- removeWhitelistSigner(...)
- addWhitelistSigner(...)
- adminVerifySignature(...)
- updateRewardTokenContract(...)
- decodeData(...)

Users: These are the wallet accounts of the users that can call the below functions:

- mint(...)
- claimReward(...)
- claimRewards(...)

4.1 General Flow

The general flow of this contract is divided into several steps. First, a manager needs to create a reward token with a specific ID through `createTokenAndDepositRewards(...)` function. A reward token is linked to rewards such as Ether, ERC20, ERC721 or ERC1155 tokens. As a second step, a minter needs to mint a reward token to a user, or a whitelisted signer has to generate a signature which will include the user's address, reward token ID and nonce—allowing users to mint their reward token through `mint(...)` function. Finally, a user can claim their reward by calling `claimReward(...)` function, which will distribute all rewards connected with the reward token.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind 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 an attacker will uncover and exploit the finding. 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.

Other factors are also considered when defining the likelihood of a finding. 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.

Other factors are also considered when defining the impact of a finding. 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 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 [High] Distribution of rewards could revert for ERC721

File(s): LootDrop.sol

Description: The function `_distributeReward(...)` distributes various rewards to a user. Rewards are divided into native Ether, `ERC20`, `ERC721` and `ERC1155` tokens.

When the `ERC721` reward type is used as part of a user's reward, it has to contain several variables. The most important is the `rewardTokenIds` array, which specifies all IDs of NFTs that will be distributed to users, and the `rewardAmount` variable, which holds the number of NFTs per reward. To track the already distributed tokens, the `LootDrop` contract contains the `erc721RewardCurrentIndex` mapping, which holds the current index of the `rewardTokenIds` array for the distribution.

The following code snippet contains a distribution of the `ERC721` reward:

```
// ...
} else if (reward.rewardType == LibItems.RewardType.ERC721) {
    uint256 currentIndex = erc721RewardCurrentIndex[_rewardTokenId][i];
    uint256[] memory tokenIds = reward.rewardTokenIds;
    for (uint256 j = 0; j < reward.rewardAmount; j++) {
        if (currentIndex >= tokenIds.length) {
            revert InsufficientBalance();
        }
         // @audit The currentIndex is not being updated
        _transferERC721(IERC721(reward.rewardTokenAddress), _from, _to, tokenIds[currentIndex]);
        erc721RewardCurrentIndex[_rewardTokenId][i]++;
    }
}
// ...
```

In the provided code, the `currentIndex` variable contains the value stored inside the `erc721RewardCurrentIndex` mapping for the current token that should be sent to the user. This variable is then used inside a loop, which iterates based on the number of tokens that should be distributed as a reward. However, the `currentIndex` variable is not updated during the loop's iterations, which causes a revert in the second iteration. As a result, the whole distribution of a reward will revert.

Recommendation(s): Consider updating the `currentIndex` variable inside the loop.

Status: Fixed

Update from the client: Fixed 1c93d95114b9edcb8681c74bcb8627eb7a4956db

6.2 [High] Possible signature replay attack across different chains

File(s): LootDrop.sol

Description: The `mint(...)` function allows users to mint a reward token or claim a reward directly. To call this function, the user must provide a `signature` created by a whitelisted signer, e.g., the protocol owner.

A signature and other user inputs are checked inside the `_verifySignature(...)` function, which is inherited from the `ERCWhitelistSignature.sol` contract.

```
function _verifySignature(
    address to,
    uint256 nonce,
    bytes calldata data,
    bytes calldata signature
) internal virtual returns (bool) {
    if (usedSignatures[signature]) revert("AlreadyUsedSignature");
    address signer = _recoverAddress(to, nonce, data, signature);
    // ...
}
```

The parameters utilized in signature validation include the recipient's address (`to`), a unique `nonce` distinguishing between various signatures, and a `data` field containing the list of authorized reward token IDs.

The `_verifySignature(...)` function effectively mitigates signature reuse by maintaining a mapping of utilized signatures, incorporating the nonce passed as a parameter. It mitigates potential signature replay attacks within the same contract instance. However, the user might still use the same signature across different instances of this contract in different EVM blockchains.

Since the intention is to implement the protocol across various blockchains, it becomes paramount to incorporate metadata within the signature to prevent this type of attack between contract instances across distinct blockchains. This can be achieved by including identifiers specific to the underlying blockchain, such as the chain ID, within the signature.

A missing chain identifier inside a signature allows an attacker to claim or mint his rewards n-times, depending on the number of EVM chains.

Recommendation(s): Consider using a chain-specific signing scheme such as EIP-155, which includes the `chain ID` in the signed message.

Status: Fixed

Update from the client: Fixed e8f407ee4c998ba2174b92683a93bc687f0a3d8c

6.3 [Medium] Lack of ERC20 transfer safety checks in LootDrop contract

File(s): LootDrop.sol

Description: The `LootDrop` contract incorporates ERC20 tokens as rewards. The manager account approves the contract to transfer the required tokens when creating a token reward by invoking the `createTokenAndDepositRewards(...)` or `createMultipleTokensAndDepositRewards(...)` functions, which internally call the `_createTokenAndDepositRewards(...)` function. This function transfers the tokens to the contract. Similarly, when a user claims a reward, the `_transferERC20(...)` function is internally called to transfer the tokens to the user.

The problem with the current implementation is that it does not handle tokens that don't comply with the ERC20 standard, e.g., tokens that don't revert on failure. In case of a silent failure of the transfer function, the reward claimers may be unable to receive their rewards.

```
// @audit called to claim user's rewards
function _transferERC20(IERC20 _token, address _to, uint256 _amount) private {
  // ...
  _token.transfer(_to, _amount);
}

// @audit called to deposit rewards
function _createTokenAndDepositRewards(...) private {
    //...
    if (reward.rewardType == LibItems.RewardType.ERC20) {
        IERC20 token = IERC20(reward.rewardTokenAddress);
        token.transferFrom(_from, _to, reward.rewardAmount);
    }
    // ...
}
```

Recommendation(s): Utilize OpenZeppelin's SafeERC20 versions with the `safeTransfer(...)` and `safeTransferFrom(...)` functions that handle the edge cases when dealing with non-ERC20 compliant tokens.

Status: Fixed

Update from the client: Fixed 1c93d95114b9edcb8681c74bcb8627eb7a4956db

6.4 [Medium] User could claim rewards even if contract is paused

File(s): LootDrop.sol

Description: The LootDrop contract has a standard Openzeppelin library Pause mechanism. However, this mechanism isn't implemented in the functions for claiming rewards, leading to an eventual unauthorized claim of rewards when the contract is in a paused state.

```
// @audit The whenNotPaused modifier is missing
function claimReward(uint256 _tokenId) external nonReentrant {
    // ...
}

function claimRewards(uint256[] calldata _tokenIds) external nonReentrant {
    // ...
}
```

Recommendation(s): Consider adding the whenNotPaused modifiers to the claim reward functions.

Status: Fixed

Update from the client: Fixed 1c93d95114b9edcb8681c74bcb8627eb7a4956db

6.5 [Low] Centralization risks

File(s): LootDrop.sol

Description: In the LootDrop contract, a single account has the authority to perform several critical operations. The _devWallet address is assigned all privileged roles: DEFAULT_ADMIN_ROLE, DEV_CONFIG_ROLE, MINTER_ROLE, MANAGER_ROLE. This address is also registered as a whitelisted signer.

Having one account performing all the critical operations introduces a significant security risk if access to this account is lost or compromised. If an attacker gains control of the _devWallet address, the contract's assets could be drained through the withdrawAssets(...) function.

Recommendation(s): To minimize security risks connected with centralization, consider distributing authority roles among different accounts and implementing multi-signature authorization for these accounts.

Status: Partially Fixed

Update from the client: Fixed 1c93d95114b9edcb8681c74bcb8627eb7a4956db

Update from Nethermind: Even though privileges were divided for multiple addresses, the centralization risks persist due to the protocol's design. We suggest implementing multi-signature wallets to minimize these risks.

6.6 [Low] itemIds can be duplicated in createTokenAndDepositRewards(...)

File(s): LootDrop.sol

Description: The _createTokenAndDepositRewards(...) function does not check if the token ID was already set, as presented in the function snippet below:

```
function _createTokenAndDepositRewards(LibItems.RewardToken calldata _token) private {
    _validateTokenInputs(_token);
    // @audit There is no check if the token ID was already set.
    tokenRewards[_token.tokenId] = _token;
    tokenExists[_token.tokenId] = true;
    itemIds.push(_token.tokenId);
    LibItems.TokenCreate memory tokenCreate = LibItems.TokenCreate(_token.tokenId, _token.tokenUri);
    rewardTokenContract.addNewToken(_token.tokenId);
    // ...
}
```

This allows for calling that function with the same token ID multiple times, in effect overriding the reward in the tokenRewards mapping and pushing duplicates of the same ID to the itemIds array. Moreover, the itemIds inside AdminERC1155Soulbound.sol could also contain multiple IDs due to invoking the addNewToken(...) function.

Recommendation(s): Consider checking if the token ID was already set to mitigate rewards overwriting and storing ID duplicates.

Status: Fixed

Update from the client: Fixed 1c93d95114b9edcb8681c74bcb8627eb7a4956db

6.7 [Info] ERC1155 with token ID zero can't be added as a reward

File(s): LootDrop.sol

Description: LootDrop allows users to earn rewards in various forms, including ERC20 tokens, ERC721, ERC1155 items, and Ether, based on predefined allocations. A manager account sets these allocations through the createTokenAndDepositRewards(...) or createMultipleTokensAndDepositRe functions, which creates the token rewards and deposits them accordingly. These functions rely on _validateTokenInputs(...) to ensure the input parameters provided by the manager are valid.

The _validateTokenInputs(...) validates different reward properties based on their type. For example, for the ERC1155 token, specifying a token ID zero is forbidden.

```
function _validateTokenInputs(LibItems.RewardToken calldata _token) private pure {
  // ...
  if (reward.rewardType == LibItems.RewardType.ERC1155) {
      // @audit ERC1155 tokens might start with token ID 0
      if (reward.rewardTokenId == 0) {
          revert InvalidTokenId();
      }
  }
  // ...
}
```

However, the ERC1155 standard allows a token ID to be set to zero. Therefore, LootDrop unnecessarily forbids some ERC1155 collections from being distributed as a reward.

Recommendation(s): Consider allowing a ERC1155 reward with token ID zero.

Status: Fixed

Update from the client: Fixed, I just removed the check since there is no need here 1c93d95114b9edcb8681c74bcb8627eb7a4956db

6.8 [Best Practices] Missing input validations

File(s): LootDrop.sol

Description: Several parts of the code do not implement proper input validation checks. To mitigate human errors, it is considered a best practice to perform input validation to ensure that the contract is always in a proper state.

- In the withdrawAssets(...) and adminBatchMintById(...) functions, the arrays received as parameters are iterated using the same index. Thus, if these arrays possess differing lengths, it may result in an out-of-bounds access. To prevent transactions from reverting, consider implementing checks to ensure that the arrays are not empty and have the same lengths. The checks are especially important when dealing with ERC1155;
- The LootDrop contract's constructor(...) does not perform the address(0) check on the _devWallet. A similar check is already present in the initialize(...) function;
- Similar concerns arise with the lack of address(0) check on the address parameter in the minting functions, e.g., mint(...), adminMint(...), adminMintById(...), and adminBatchMintById(...). Given that all these functions invoke the internal function _mintAndClaimRewardToken(...), consider adding the validation within this function;

Recommendation(s): Consider validating the inputs according to the above recommendations.

Status: Fixed

Update from the client: Fixed 1c93d95114b9edcb8681c74bcb8627eb7a4956db

6.9 [Best Practices] Perform input parameter validation before modifying the contract state

File(s): LootDrop.sol

Description: In some functions, input validation checks are performed after a gas-expensive operation such as writing to storage. This may result in an unnecessary gas expense in case of a transaction failure due to a check reverting.

For instance, within the `_mintAndClaimRewardToken(...)` function, the execution gas could be saved if the `_amount` check would be performed before writing to storage.

```
function _mintAndClaimRewardToken(...) private {
    // ...
    // @audit Perform the check before writing to storage.
    currentRewardSupply[_tokenId] += _amount;
    // ...
    // @audit Perform this check earlier to follow the CEI pattern and save gas.
    if (_amount == 0) {
        revert InvalidAmount();
    }
    // ...
}
```

Recommendation(s): Consider applying the Check-Effect Interaction pattern to make the code more robust while saving gas in the process.

Status: Fixed

Update from the client: `_mintAndClaimRewardToken()` fixed, not sure anywhere else in the contract has this issue. Can you point it out for me please? 1c93d95114b9edcb8681c74bcb8627eb7a4956db

6.10 [Best Practices] Unused Code

File(s): LootDrop.sol

Description: The codebase contains some unused code, affecting overall readability and code quality. A non-exhaustive list of unused declarations is shown below:

- `tokenCreate` variable in the `_createTokenAndDepositRewards(...)` function;
- `tokenIdProcessed` mapping state variable;
- `TokenUpdated` event;

Recommendation(s): To keep the code clean and readable, consider removing unused functionality.

Status: Fixed

Update from the client: Fixed 1c93d95114b9edcb8681c74bcb8627eb7a4956db

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. 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 Summon documentation

The `Summon` team has described the contract during the code walk-through. Additionally, supporting diagrams of interaction flows were provided. Unfortunately, the codebase lacks detailed comments on individual functions.

8 Test Suite Evaluation

8.1 Compilation Output

```
> forge compile
[] Compiling...
[] Compiling 177 files with 0.8.17
[] Solc 0.8.17 finished in 436.91s
Compiler run successful with several warnings:

// ...
```

8.2 Tests Output

```
> forge test

Running 12 tests for test/LootDropHQ/LootDropHQWithdraw.t.sol:LootDropWithdrawTest
[PASS] testWithdrawAddressZeroShouldFail() (gas: 13536)
[PASS] testWithdrawERC1155NotOwnedShouldFail() (gas: 28436)
[PASS] testWithdrawERC1155ShouldPass() (gas: 131545)
[PASS] testWithdrawERC1155TooMuchShouldFail() (gas: 76425)
[PASS] testWithdrawERC20ShouldPass() (gas: 51788)
[PASS] testWithdrawERC20TooMuchShouldFail() (gas: 24350)
[PASS] testWithdrawERC721InvalidTokenIdShouldFail() (gas: 27590)
[PASS] testWithdrawERC721NotOwnedShouldFail() (gas: 84846)
[PASS] testWithdrawERC721ShouldPass() (gas: 61625)
[PASS] testWithdrawETHShouldPass() (gas: 47368)
[PASS] testWithdrawETHTooMuchShouldFail() (gas: 16151)
[PASS] testWithdrawNotManagerRoleShouldFail() (gas: 39143)
Test result: ok. 12 passed; 0 failed; 0 skipped; finished in 9.65ms

Running 8 tests for test/LootDropHQ/LootDropHQClaim.t.sol:LootDropClaimTest
[PASS] testAdminClaimRewardClaimPausedShouldFail() (gas: 341678)
[PASS] testAdminClaimRewardShouldPass() (gas: 331850)
[PASS] testAdminMintAndClaimMultipleERC721ShouldPass() (gas: 1650550)
[PASS] testAdminMintAndClaimNotEnoughETHShouldFail() (gas: 204132)
[PASS] testAdminMintAndClaimShouldPass() (gas: 573603)
[PASS] testClaimRewardShouldPass() (gas: 334861)
[PASS] testUserClaimMultipleRewardsERC721AllGoneShouldFail() (gas: 397531)
[PASS] testUserClaimMultipleRewardsShouldPass() (gas: 370523)
Test result: ok. 8 passed; 0 failed; 0 skipped; finished in 9.14ms

Running 5 tests for test/LootDropHQ/LootDropHQBurn.t.sol:LootDropBurnTest
[PASS] testBurn() (gas: 752644)
[PASS] testBurnBatch() (gas: 785987)
[PASS] testBurnBatchNotOwnerShouldFail() (gas: 446354)
[PASS] testBurnIfHoldBothNonSoulboundAndSouldbound() (gas: 558492)
[PASS] testBurnNotOwnerShouldFail() (gas: 361467)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 222.82ms

Running 9 tests for test/LootDropHQ/LootDropHQAddToken.t.sol:LootDropAddTokenTest
[PASS] testAddMultipleRewardTokensShouldPass() (gas: 1777134)
[PASS] testAddNewTokensNotDEV_CONFIG_ROLEShouldFail() (gas: 277940)
[PASS] testAddNewTokensNotEnoughERC1155ShouldFail() (gas: 983996)
[PASS] testAddNewTokensNotEnoughERC20ShouldFail() (gas: 812713)
[PASS] testAddNewTokensNotEnoughETHShouldFail() (gas: 534460)
[PASS] testAddNewTokensShouldPass() (gas: 1022107)
[PASS] testAddNtestAddNewTokensDontOwnedERC1155ShouldFail() (gas: 985269)
[PASS] testAddNtestAddNewTokensNotEnoughERC721ShouldFail() (gas: 952392)
[PASS] testTokenExists() (gas: 1413986)
Test result: ok. 9 passed; 0 failed; 0 skipped; finished in 243.56ms
```

```
Running 9 tests for test/LootDropHQ/LootDropHQ.t.sol:LootDropTest
[PASS] testDecodeDataShouldPass() (gas: 623221)
[PASS] testInitializeTwiceShouldFail() (gas: 14318)
[PASS] testInvalidSignature() (gas: 55390)
[PASS] testPauseUnpause() (gas: 154686)
[PASS] testPauseUnpauseSpecificToken() (gas: 485100)
[PASS] testReuseSignatureMint() (gas: 412618)
[PASS] testUpdateRewardTokenContractAddressZeroShouldFail() (gas: 12255)
[PASS] testUpdateRewardTokenContractNotAuthorizedShouldFail() (gas: 39510)
[PASS] testUpdateRewardTokenContractShouldPass() (gas: 15776)
Test result: ok. 9 passed; 0 failed; 0 skipped; finished in 243.96ms

Running 7 tests for test/LootDropHQ/LootDropHQMint.t.sol:LootDropMintTest
[PASS] testAdminMint() (gas: 378355)
[PASS] testAdminMintNotMinterRole() (gas: 55682)
[PASS] testMintExceedSupplyShouldFail() (gas: 970758)
[PASS] testMintInvalidTokenId() (gas: 62055)
[PASS] testMintShouldPass() (gas: 763560)
[PASS] testadminMintById() (gas: 137874)
[PASS] testadminMintByIdNotMinterRole() (gas: 43265)
Test result: ok. 7 passed; 0 failed; 0 skipped; finished in 244.15ms

Running 4 tests for test/LootDropHQ/LootDropHQTransfer.t.sol:LootDropTransferTest
[PASS] testBatchTransferFrom() (gas: 604829)
[PASS] testNonSoulboundTokenTransfer() (gas: 137994)
[PASS] testSoulboundTokenNotTransfer() (gas: 153571)
[PASS] testSoulboundTokenTransferOnlyWhitelistAddresses() (gas: 228102)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 248.70ms

Ran 7 test suites: 54 tests passed, 0 failed, 0 skipped (54 total tests)
```

Remarks about Summon test suite

The **Summon** team has diligently crafted test suites that encapsulate the fundamental workflows for interacting with the protocol. However, the current test suite lacks comprehensiveness, especially regarding edge cases and functionalities. For example, tests for scenarios with multiple claimants and accounting validation still need to be included.

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 procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.