Security Review Report
NM-0151 Libertas Omnibus

NETHERMIND
SECURITY

(Dec 20, 2023)

Contents

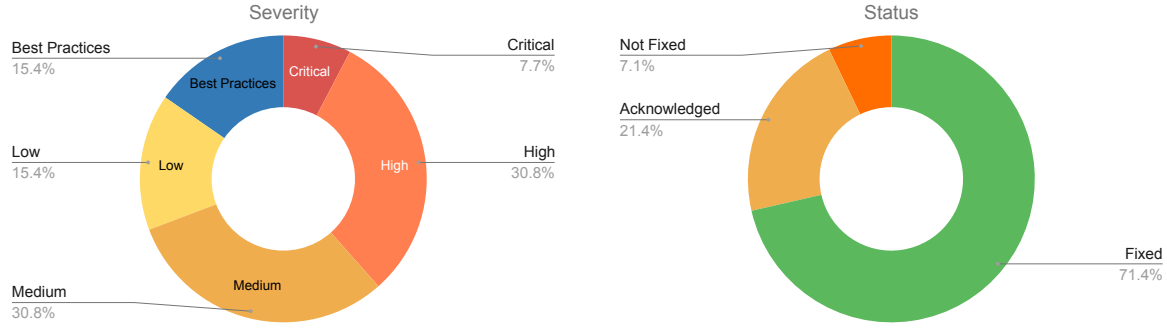
1 Executive Summary

This document outlines the security review conducted by Nethermind for the Game7. Libertas Omnibus is an experimental NFT project. It allows for minting soulbound tokens that can be used in the gaming experience.

The audited code comprises 1,097 lines of Solidity code. In the course of the audit, it was observed that the codebase lacks comprehensive testing of its primary features. This resulted in our team identifying a critical issue that could have been uncovered through more extensive testing of the core project's functionalities. Consequently, we recommend that the team meticulously assess the entire codebase before proceeding with deployment. The client was unable to furnish a Git repository containing a history of changes and commit hashes. Consequently, instead of presenting a commit hash indicating the code version, we are incorporating `sha256` hashes of the audited files. The hashes were created using the command `openssl sha256 <file>`.

The audit was performed using: (a) manual analysis of the codebase and (b) simulation of the smart contracts. **Along this document, we report** 14 points of attention, where one is classified as `Critical`, four are classified as `High`, four are classified as `Low`, five 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 (1), **High** (4), **Medium** (0), **Low** (4), **Undetermined** (0), **Informational** (2), **Best Practices** (3). **(b) Distribution of status: Fixed** (0), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	December 20, 2023
Response from Client	-
Final Report	-
Methods	Manual Review, Automated Analysis, Tests
Repository	Not available
Commit Hash	Not available
Documentation	Code comments
Documentation Assessment	Medium
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	ERCSoulbound.sol	133	82	61.7%	16	231
2	Soulbound1155.sol	213	27	12.7%	49	289
3	ItemBound.sol	255	24	9.4%	59	338
4	ERCWhitelistSignature.sol	34	25	73.5%	11	70
5	upgradeables/ERCWhitelistSignatureUpgradeable.sol	39	26	66.7%	13	78
6	upgradeables/ERCSoulboundUpgradeable.sol	118	83	70.3%	18	219
7	upgradeables/AvatarBoundV1.sol	283	27	9.5%	47	357
8	libraries/LibItems.sol	22	1	4.5%	3	26
	Total	**1097**	**295**	**26.9%**	**216**	**1608**

Hashes of audited files:

	Contract	SHA2-256(Contract File)
1	ERCSoulbound.sol	b1bd7de0202fb3f0a35a19d6e4b6dcba6c8738db27e963134661e2aa8202838f
2	Soulbound1155.sol	953c86f92c3135f3140a05962698e51f7910acfb27cd56595091bf40b813c474
3	ItemBound.sol	54cb1c5d23cf4e4b4d1af20d96317a5acd1ffcc8fe5307f0194c594a42c8f05b
4	ERCWhitelistSignature.sol	93f9f0111249ac8fa235c7bc069353c75304f2a1505a7cd3bf0e92b64d2be531
5	ERCWhitelistSignatureUpgradeable.sol	3f766b92bd1cf8a8909ef8c5d72b63fcb16e4541fee00107d4650ae24539f120
6	ERCSoulboundUpgradeable.sol	5e630d8130dbce0ec16b8f5fbc28f6faf83a2719885b7bfb0b0afa897e25ec01
7	AvatarBoundV1.sol	cb382ed9b63bdaf5063c1683b1d910345f51d2ffaaa0e01ec0d9fe3d71223dee
8	LibItems.sol	161578685be66813c80b90bb1841279aaf0ca2ceed3385b1d2a6514af65589e0

Hashes of files with fixes:

	Contract	SHA2-256(Contract File)
1	ERC1155Soulbound.sol	d099674b23ec5452708e03dbc01f7e887463e66a1d9513dd3df9c7ace24274d3
2	ItemBound.sol	0c1f1020be473a56076de6eff600755d63b085bde87161d25e207066a113df66
3	ERCWhitelistSignature.sol	2b0658fdf129519e0b4a968ec27d190a56539cb4ac6d5802523182f3c83354c2
4	ERCWhitelistSignatureUpgradeable.sol	deb3931bd50ee5a24417b72ec4e9d9140f67a53d61c7f5b1139a15cd6c8b417e
5	ERC721SoulboundUpgradeable.sol	97b311330d4fdab1a91ee7c9eb1f3bc28cd019a96d1b5ea066e04a812958b712
6	AvatarBoundV1.sol	4e2b1a3d388a39bd7cd4e1ea53374f73822b39dd64c389870a932ecb33ac9630
7	LibItems.sol	9560a3227efa747fe27937459e9832e4b35bf7835b822d322414fc5330ca10b2

3 Summary of Issues

	Finding	Severity	Update
1	The function `mintAvatarNftGating(...)` will always revert	Critical	Fixed
2	User can burn soulbound tokens with `burnBatch(...)`	High	Fixed
3	User may mint any token ID	High	Fixed
4	Incorrect check for ERC1155 soulbound tokens batch transfers	High	Fixed
5	`AvatarBoundV1` soulbound tokens transfer on mint	High	Fixed
6	Admin can mint any amount of SoulBound1155 tokens with `adminMintBatch(...)`	Low	Acknowledged
7	Centralization risks	Low	Acknowledged
8	It is possible to bypass the maximum number of items per mint by passing duplicate item IDs	Low	Acknowledged
9	Unused functions inherited in token contracts	Low	Not Fixed
10	Redundant modifiers	Info	Fixed
11	Unnecessary checks	Info	Fixed
12	Code layout does not adhere to the official Solidity Style Guidelines.	Best Practices	Fixed
13	Incorrect comment	Best Practices	Fixed
14	`__gap` variable	Best Practices	Fixed

4 System Overview

Libertas Omnibus allows users to mint tokens in `ERC721` and `ERC1155` standard. Each token is soulbound or optionally soulbound, meaning it can't be transferred or burned once minted to the given address. The `AvatarBoundV1.sol` contract is `ERC721` token that represents user's avatar. The contract uses the following storage variables:

```
uint256[50] private __gap;
bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");
uint256 private _tokenIdCounter;
uint256 public _baseSkinCounter;
uint256 private _specialItemId;
uint256 private defaultItemId;
string public baseTokenURI;
string public contractURI;
string public revealURI;
address public gatingNFTAddress;
address public itemsNFTAddress;
bool private mintNftGatingEnabled;
bool private mintNftWithoutGatingEnabled;
bool private mintRandomItemEnabled;
bool private mintSpecialItemEnabled;
bool private mintDefaultItemEnabled;
mapping(uint256 => string) public baseSkins;
```

The contract exposes following public or external functions:

Initialization function

```
function initialize(...) public;
```

Mints avatar token and additional item tokens

```
function mintAvatar(...) public;
```

Mints avatar token for the owner of gating token

```
function mintAvatarNftGating(...) public;
```

Role-restricted avatar token minting

```
function adminMint(...) public;
function batchMint(...) public;
```

Role-restricted signature utilization

```
function adminVerifySignature(...) public;
```

Role-restricted pausing functions

```
function pause(...) public;
function unpause(...) public;
```

Role restricted setters

```
function batchSetTokenURI(...) public;
function setContractURI(...) public;
function setTokenURI(...) public;
function setBaseURI(...) public;
function setRevealURI(...) public;
function setBaseSkin(...) public;
function setMintRandomItemEnabled(...) public;
function setMintDefaultItemEnabled(...) public;
function setItemsNFTAddress(...) public;
function setNftGatingAddress(...) public;
function setSpecialItemId(...) public;
function setDefaultItemId(...) public;
function setMintNftGatingEnabled(...) public;
function setMintSpecialItemEnabled(...) public;
function setMintNftWithoutGatingEnabled(...) public;
function addWhitelistSigner(...) external;
function removeWhitelistSigner(...) external;
```

View functions

```
function getAllBaseSkins(...) public;
function tokenURI(...) public;
```

The contract inherits the following OpenZeppelin contracts:

```
Initializable,
ERC721EnumerableUpgradeable,
ERC721URIStorageUpgradeable,
AccessControlUpgradeable,
PausableUpgradeable,
ReentrancyGuardUpgradeable
```

And the following custom contracts:

```
ERCSoulboundUpgradeable,
ERCWhitelistSignatureUpgradeable
```

The `ItemBound.sol` contract is `ERC1155` token that represents user's item. The contract uses the following storage variables:

```
 string public contractURI;
string private baseURI;
string public name;
string public symbol;
using Strings for uint256;
uint256 public currentMaxLevel;
uint256 public MAX_PER_MINT;
mapping(uint256 => bool) public tokenExists;
mapping(uint256 => string) public tokenUris;
mapping(uint256 => bool) public isTokenMintPaused;
mapping(LibItems.Tier => mapping(uint256 => uint256[])) public itemPerTierPerLevel;
uint256[] public itemIds;
```

The contract exposes the following public or external functions:

Initialization function

```
function constructor(...);
```

Mints item tokens

```
function mint(...) external;
```

Burn item tokens

```
function burn(...) public;
function burnBatch(...) public;
```

Role-restricted item token minting

```
function adminMint(...) external;
function adminMintId(...) external;
```

Role-restricted signature utilization

```
function adminVerifySignature(...) public;
```

Role-restricted pausing functions

```
function pause(...) public;
function unpause(...) public;
```

Role-restricted setters

```
function addNewToken(...) public;
function addNewTokens(...) public;
function updateTokenUri(...) public;
function updateTokenMintPaused(...) public;
function updateBaseUri(...) external;
function setRoyaltyInfo(...) external;
function updateWhitelistAddress(...) public;
function setContractURI(...) public;
function addWhitelistSigner(...) external;
function removeWhitelistSigner(...) external;
```

View functions

```
function getAllItems(...) public;
function isTokenExist(...) public;
function getCurrentMaxLevel(...) public;
function getItemsPerTierPerLevel(...) public;
function balanceOfBatchOneAccount(...) public;
function uri(...) public;
```

The contract inherits the following OpenZeppelin contracts:

```
ERC1155Burnable,
ERC1155Supply,
ERC2981,
AccessControl,
Pausable,
ReentrancyGuard
```

and following custom contracts:

```
ERCSoulbound,
ERCWhitelistSignature
```

The `Soulbound1155.sol` contract is an `ERC1155` token that represents the users' items. The token has been removed from the final version of the code.

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.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind 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 formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] The function `mintAvatarNftGating(...)` will always revert

File(s): `AvatarBoundV1.sol`

Description: The function `mintAvatarNftGating(...)` will mint AvatarBound NFT to the caller who owns NFT gating. However, even though the caller meets the requirements above, the function will still revert because the function `revealNFTGatingToken(...)`. This function implements `onlyRole(MINTER_ROLE)`, which checks if the caller has this role. This role most likely won't be granted to every user calling `mintAvatarNftGating(...)`. Therefore, the function will revert as a result of the missing role.

```
1   function revealNFTGatingToken(uint256 tokenId) public onlyRole(MINTER_ROLE) whenNotPaused {
2       IOpenMint(gatingNFTAddress).reveal(tokenId, revealURI);
3       emit NFTRevealed(tokenId, _msgSender(), gatingNFTAddress);
4    }
```

Recommendation(s): Consider changing the access control for the `revealNFTGatingToken(...)`.

Status: Fixed

Update from the client: This function has been changed to private function and the `onlyRole()` modifier was removed.

6.2 [High] User can burn soulbound tokens with `burnBatch(...)`

File(s): `SoulBound1155.sol`, `ItemBound.sol`

Description: The holder of soulbound tokens should not be able to burn them. This property is checked in the modifiers `soulboundCheck(...)` and `soulboundCheckBatch(...)`. However, if the user holds both soulbound and non-soulbound tokens with the same `id`, it is possible to burn soulbound tokens by calling `burnBatch(...)`. Consider the following scenario:

- Bob holds amount=1 soulbound token and amount=1 non-soulbound token with id=1;
- Bob calls `burnBatch(account=Bob, tokenIds=[1, 1], amounts=[1, 1])` and burns all tokens, including soulbound one;

This issue is possible because `soulboundCheckBatch(...)` calls in a loop `_checkMultipleAmounts(...)`, which effectively performs the same checks multiple times. This allows the user to burn multiple tokens with the same `id` by burning them in a batch. Note that this issue exists in both `SoulBound1155` and `ItemBound` contracts. Refer to the finding 6.4 for an in-depth explanation.

Recommendation(s): Ensure that the soulbound tokens can't be burned in both `Soulbound1155` and `ItemBound` token contracts.

Status: Fixed

Update from the Nethermind: The issue was present in `Soulbound1155.sol`, which was removed and is not present in the final version.

6.3 [High] User may mint any token ID

File(s): `Soulbound1155.sol`

Description: The signed message required to mint tokens contains only the address and nonce but does not contain the `id`. Therefore, the provided token `id` is arbitrary. This may result in:

- minting token by a user that should not be allowed to mint;
- inconsistency between off-chain and on-chain data;

Recommendation(s): Consider adding token `id` in the signed message.

Status: Fixed

Update from the Nethermind: The issue was present in `Soulbound1155.sol`, which was removed and is not present in the final version.

6.4 [High] Incorrect check for ERC1155 soulbound tokens batch transfers

File(s): `SoulBound1155.sol`, `ItemBound.sol`

Description: The `SoulBound1155` and `ItemBound` contracts mint ERC1155 tokens. The tokens can be soulbounded to a specific user address during minting, meaning those users should not be able to transfer those tokens. The same tokens can be minted to another user as non-soulbound tokens, meaning that this user can transfer tokens. One address may hold both soulbound tokens and non-soulbound tokens. Whether the token is soulbounded to a specific user address is defined during mint by the parameter `soulBound` and registered with the function `_soulbound(...)`. The ability to transfer tokens is checked during the transfer in the modifier `soulboundCheck(...)` and `soulboundCheckBatch(...)`. Another set of modifiers updates the state of soulbound tokens: `syncSoulbound(...)` and `syncBatchSoulbound(...)`. However, those checks do not work properly during batched transfer with `safeBatchTransferFrom(...)`, allowing the user to transfer soulbound tokens. Consider the following scenario:

- Bob holds one soulbound token id=1;
- Alice holds one non-soulbound token id=1;
- Alice transfers one token id=1 to Bob;
- Bob holds one soulbounded token id=1 and one non-soulbounded token id=1;
- Bob transfers two tokens id=1 (including the soulbound one) to Eve by calling:;

```
safeBatchTransferFrom(from=Bob, to=Eve, tokenIds=[1, 1], amounts=[1, 1], data="")
```

- Eve now holds two non-soulbound tokens id=1;

The presented scenario allows for breaking the core functionality of soulbound tokens, i.e., non-transferability. The issue exists because of ineffective checks in `soulboundCheckBatch(...)` presented below:

```
modifier soulboundCheckBatch(
    address from,
    address to,
    uint256[] memory tokenIds,
    uint256[] memory amounts,
    uint256[] memory totalAmounts
) {
    require(tokenIds.length == amounts.length, "ERCSoulbound: tokenIds and amounts length mismatch");
    for (uint256 i = 0; i < tokenIds.length; i++) {
        _checkMultipleAmounts(from, to, tokenIds[i], amounts[i], totalAmounts[i]);
    }
    _;
}
```

The modifier `soulboundCheckBatch(...)` calls in a loop `_checkMultipleAmounts(...)`:

```
function _checkMultipleAmounts(address from, address to, uint256 tokenId, uint256 amount, uint256 totalAmount) private
↪ view {
    require(from != address(0), "ERCSoulbound: can't be zero address");
    require(amount > 0, "ERCSoulbound: can't be zero amount");
    require(amount <= totalAmount, "ERCSoulbound: can't transfer more than you have");
    // check if from or to whitelist addresses let it through
    if (whitelistAddresses[from] || whitelistAddresses[to]) {
        return;
    }
    if (totalAmount - _soulbounds[from][tokenId] < amount) {
        revert("ERCSoulbound: The amount of soulbounded tokens is more than the amount of tokens to be transferred");
    }
}
```

The `_checkMultipleAmounts(...)` checks if the transferred tokens include only non-soulbound tokens. However, during the batch transfer with `safeBatchTransferFrom(...)`, malicious user may provide the same token multiple times (e.g., ids=[1,1], amounts=[1,1]), and the check presented in the function `_checkMultipleAmounts(...)` are performed on the same state, effectively checking multiple times the same thing, i.e., whether the user can send amount=1 of token id=1. After the check, the transfers are performed, and the user sends amount=2 of token id=1, where one token is non-soulbound. Additionally, the update of the balance of soulbound tokens performed in `syncBatchSoulbound(...)` will update only the malicious sender's balance of soulbound tokens, and the receiver receives non-soulbound tokens.

Recommendation(s): Ensure that the soulbound tokens are non-transferrable in both `Soulbound1155` and `ItemBound` token contracts.

Status: Fixed

Update from the Nethermind: The issue was present in `Soulbound1155.sol`, which was removed and is not present in the final version.

6.5 [High] `AvatarBoundV1` soulbound tokens transfer on mint

File(s): `AvatarBoundV1.sol`

Description: The `AvatarBoundV1` is an ERC712 soulbound token. The address for which this token is minted should not be able to transfer it. However, the malicious receiver, during the safe mint, may reenter the `AvatarBoundV1` and call `AvatarBoundV1.transferFrom(...)` to transfer the token to another address (A). This not only breaks the soulbound property of the receiver but also, the new address A is able to transfer the token further. The issue exists because the function `mint(...)` calls inherited from `ERC721Upgradeable` function `_safeMint(...)`, presented below:

```
function _safeMint(address to, uint256 tokenId, bytes memory data) internal virtual {
    _mint(to, tokenId);
    require(
        _checkOnERC721Received(address(0), to, tokenId, data),
        "ERC721: transfer to non ERC721Receiver implementer"
    );
}
```

This function checks if the receiver is correct ERC721Receiver with `_checkOnERC721Received(...)`:

```
function _checkOnERC721Received(
    address from,
    address to,
    uint256 tokenId,
    bytes memory data
) private returns (bool) {
    if (to.isContract()) {
        try IERC721ReceiverUpgradeable(to).onERC721Received(_msgSender(), from, tokenId, data) returns (bytes4 retval) {
            return retval == IERC721ReceiverUpgradeable.onERC721Received.selector;
        } catch (bytes memory reason) {
            if (reason.length == 0) {
                revert("ERC721: transfer to non ERC721Receiver implementer");
            } else {
                /// @solidity memory-safe-assembly
                assembly {
                    revert(add(32, reason), mload(reason))
                }
            }
        }
    } else {
        return true;
    }
}
```

The check is done after the mint by performing the external call to the receiver with `IERC721ReceiverUpgradeable(to).onERC721Received(...)`. The malicious receiver may be a contract, which reenters the `AvatarBoundV1` to transfer the token to another address. Note that currently the `AvatarBoundV1` prohibits any token transfers by overriding `_beforeTokenTransfer(...)` with modifier `soulboundAddressCheck(from)`:

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 tokenId,
    uint256 batch
) internal override(ERC721Upgradeable, ERC721EnumerableUpgradeable) soulboundAddressCheck(from) {
    super._beforeTokenTransfer(from, to, tokenId, batch);
}
```

However, this check does not fix the presented scenario since the address is registered as soulbound after the `_safeMint(...)` call. Therefore, the transfer during mint is still possible.

Recommendation(s): If the `AvatarBoundV1` tokens should not be transferrable, consider disabling transfer functions instead of adding a modifier to the `_beforeTokenTransfer(...)` hook.

Status: Fixed

Update from the Nethermind: The core issue was fixed by disabling functions: `transferFrom(...)` and `safeTransferFrom(...)`. However, currently the modifier `soulboundAddressCheck(...)` added to `_beforeTokenTransfer(...)` is unnecessary. Also, both functions `safeTransferFrom(...)` contain unreachable code, which can be removed:

```
function safeTransferFrom(
    address from,
    address to,
    uint256 tokenId
) public override(IERC721Upgradeable, ERC721Upgradeable) nonReentrant {
    revert("You can't transfer this token");
    ////////////////////
    // @audit: unreachable code
    ////////////////////
    super.safeTransferFrom(from, to, tokenId, "");
}
function safeTransferFrom(
    address from,
    address to,
    uint256 tokenId,
    bytes memory data
) public override(IERC721Upgradeable, ERC721Upgradeable) nonReentrant {
    revert("You can't transfer this token");
    ////////////////////
    // @audit: unreachable code
    ////////////////////
    super._safeTransfer(from, to, tokenId, data);
}
```

6.6 [Low] Admin can mint any amount of SoulBound1155 tokens with `adminMintBatch(...)`

File(s): `SoulBound1155.sol`

Description: The caller of the `SoulBound1155.adminMintBatch(...)` is able to mint any arbitrary amount of tokens by passing to repeated values in the arrays `ids` and `amounts`. Below, we present the modifier `canMintBatch(...)` that checks if the provided amount is correct:

```
modifier canMintBatch(
    address to,
    uint256[] memory tokenIds,
    uint256[] memory amounts
) {
    for (uint256 i = 0; i < tokenIds.length; i++) {
        if (!tokenExists[tokenIds[i]]) {
            revert("Token not exist");
        }
        if (isMinted[tokenIds[i]][to]) {
            revert("Already minted");
        }
        if (amounts[i] > MAX_PER_MINT) {
            revert("Exceed max mint");
        }
    }
    _;
}
```

This modifier will iterate through each element of both arrays and check if the token was already minted to the given address and if the amount is less than `MAX_PER_MINT`. If provided arrays are, e.g., `ids=[1,1,1]`, `amounts=[1,1,1]`, then the checks in the `canMintBatch(...)` modifier will pass since the token with `id=1` was not yet minted, and each value in `amounts` is correct. Those arrays are used in `ERC1155._mintBatch(...)`, which, as a result, mints to the given address `amount=3` of token `id=1`. Note that this issue is called only by the `MINTER_ROLE`, but allows for breaking the assumption of minting `amount=1` of tokens.

Recommendation(s): Consider restricting duplicates in modifier `canMintBatch(...)`.

Status: Acknowledged

Update from the client: We're deleting the modifier because this function is for admin purposes.

6.7 [Low] Centralization risks

File(s): `AvatarBoundV1.sol`, `Soulbound1155.sol`, `ItemBound.sol`

Description: The Libertas Omnibus project has centralized points. Centralization risks are connected with scenarios that can negatively impact protocol if the private keys of admins or privileged roles are compromised. The risks include the following areas:

- direct, one-step role granting;
- functions accessible by privileged roles have weakened security checks than the public functions;
- some aspects of token mint security are ensured by the correct data signing by the backend;

Recommendation(s): Consider implementing stronger checks to increase the security of centralized points in the application.

Status: Acknowledged

Update from the client: We're aware of this. We'll be sharing this info with both teams to publish the comms.

6.8 [Low] It is possible to bypass the maximum number of items per mint by passing duplicate item IDs

File(s): `ItemBound.sol`

Description: The `mint(...)` function is used by users to mint the items for their Avatars. It validates the user's signature and the `amount` of items that the user wants to mint, not exceeding the `MAX_PER_MINT` amount. The information about which item tokens should be minted is passed in the `data` argument. The problem present in the `mint(...)` function is that this data can contain duplicate values. Under the assumption that the project's backend does not check the uniqueness of items to be minted, it is possible for the user to specify the same item multiple times in the `data` argument and, as a result, mint one item more than once, exceeding the `MAX_PER_MINT` amount.

Recommendation(s): One of the solutions would be to mint a single token at a time instead of using the `_mintBatch(...)` function. If that's not possible, consider implementing a check on a smart contract level to ensure that the user's item data contains unique values to prevent the minting of the same item multiple times.

Status: Acknowledged

Update from the client: This is by design. Users will be able to get duplicate items.

6.9 [Low] Unused functions inherited in token contracts

File(s): `ERCSoulBound.sol`, `ERCSoulboundUpgradeable.sol`

Description: Some functions are inherited from the contracts `ERCSoulBound` and `ERCSoulboundUpgradeable` but are not used. This may mislead users, as the functions called return incorrect information. Unused functions also unnecessarily increase contract size. Below, we list unused functions that are inherited. `AvatarBoundV1` inherits from `ERCSoulboundUpgradeable` but does not use:

- modifier `soulboundTokenCheck(...)`;
- modifier `soulboundCheck(...)`;
- modifier `soulboundCheckBatch(...)`;
- modifier `syncSoulboundToken(...)`;
- modifier `syncSoulbound(...)`;
- modifier `syncBatchSoulbound(...)`;
- modifier `revertOperation(...)`;
- internal function `_updateWhitelistAddress(...)`;
- internal function `_checkMultipleAmounts(...)`;
- internal function `_soulboundToken(...)`;
- internal function `_soulbound(...)`;
- internal function `_soulboundBatch(...)`;
- external function `isSoulboundToken(...)` is exposed, but mapping `_soulboundTokens` is not used;
- external function `soulboundBalance(...)` is exposed, but mapping `_soulbounds` is not used;
- internal function `__ERCSoulboundUpgradable_init(...)` is used, but is empty;

`ItemBound` and `Soulbound1155` inherit from `ERCSoulbound`, but do not use:

- modifier `soulboundTokenCheck(...)`;
- modifier `soulboundAddressCheck(...)`;
- modifier `syncSoulboundToken(...)`;
- modifier `revertOperation(...)`;
- internal function `_soulboundToken(...)`;
- internal function `_soulboundAddress(...)`;
- external function `isSoulboundToken(...)` is exposed, but mapping `_soulboundTokens` is not used;
- external function `isSoulboundAddress(...)` is exposed, but mapping `_soulboundAddresses` is not used;

The external functions that the contracts inherit and expose may return misleading results to the user. Consider, e.g., call to external function `AvatarBoundV1.isSoulboundToken(...)`, which always returns `false`, but the user may expect that provided for `tokenId` it should return `true`.

Recommendation(s): Removing unused modifiers and functions from `ERCSoulBound` and `ERCSoulboundUpgradeable`.

Status: Not Fixed

Update from the Nethermind: The contracts still inherit unused modifiers/functions that include external functions, which can be misleading to the users. E.g.:

- external function `AvatarBoundV1.isSoulboundToken(...)` reads `_soulboundTokens` mapping that is never changed;
- public function `ItemBound.isSoulboundAddress(...)` reads `_soulboundAddresses` mapping that is never changed;

Update from the client: We acknowledged this, and we decided to keep this since it's a library. We will use this function in future contracts.

6.10 [Info] Redundant modifiers

File(s) AvatarBoundV1.sol

Description: The AvatarBoundV1.sol uses the `PausableUpgradeable` extension for the pausing functionality. Several internal functions use the `whenNotPaused` modifier, which is unnecessary as the public functions, which expose these internal functions, already implement this modifier. The modifier can be deleted from the following functions to reduce the gas cost:

- `revealNFTGatingToken(...);`
- `mintItem(...);`
- `mintRandomItem(...);`

Recommendation(s): Consider removing the `whenNotPaused` modifier to avoid redundant checks.

Status: Fixed

Update from the client: The modifier was removed in the abovementioned functions.

6.11 [Info] Unnecessary checks

File(s): `ERCSoulbound.sol`

Description: In several parts of the application, the smart contracts perform checks for data validation, which are unnecessary as they are implemented in the inherited functions or do not prevent any security risk. The function `_checkMultipleAmounts(...)` is called on every transfer or burn in `Soulbound1155` and `ItemBound` contracts. However, some checks in this function are unnecessary, as they are either done again in the functions inherited with ERC1155 or do not introduce additional security. Below, we list checks that are duplicated in `_checkMultipleAmounts(...)`, with corresponding reasoning:

- `require(amount <= totalAmount, "ERCSoulbound: can't transfer more than you have")` - the check is done again in `ERC1155._safeTransferFrom(...)/_safeBatchTransferFrom(...)` and `ERC1155._burn(...)/_burnBatch(...)`;
- `require(from != address(0), "ERCSoulbound: can't be zero address")` - the check is done again in `ERC1155._burn(...)/_burnBatch(...)`. Note, however, that this is not checked during transfer in `ERC1155`;

Additionally, the `Soulbound1155.__mintBatch(...)` calls the `_soulboundBatch(...)`. This function performs check `require(tokenIds.length == amounts. length, "ERCSoulbound: tokenIds and amounts length mismatch")` which verifies the length equality of arrays. However, the same check is repeated in the `_mintBatch(...)` function inherited from `ERC1155`, which is also called in `Soulbound1155.__mintBatch(...)`.

Recommendation(s): Consider avoiding double-checks to simplify the code and save gas.

Status: Fixed

Update from the client: Removed those checks.

Update from the Nethermind: Note that the check `require(from != address(0), "ERCSoulbound: can't be zero address")` now is not done for the transfer functions.

6.12 [Best Practice] The code layout does not adhere to the official Solidity Style Guidelines.

File(s): `*`

Description: Solidity language has a set of general rules to keep the code style consistent across different projects, contracts, and functions. A non-exhaustive list of files where some of these rules are broken is presented below:

- `ItemBound.sol`: The `constructor` is defined after `public` functions;
- `AvatarBoundV1.sol`: The `constructor` is defined before state variables;
- `ERCSoulbound.sol`, `ERCSoulboundUpgradeable.sol`: The `external` functions are defined before `private` and `internal` ones;

Recommendation(s): Consider reviewing and applying the official Solidity Style Guideliness to make the code easier to navigate and comprehend for the readers.

Status: Fixed

Update from the client: Reorganized all of our contracts using the guidelines.

6.13 [Best Practice] Incorrect comment

File(s): `SoulBound1155.sol`

Description: The comment at the beginning of the contract states that:

```
1 ///@notice This contract is for mock for WETH token.
```

Recommendation(s): Consider removing the misleading comment.

Status: Fixed

Update from the client: We're deleting this contract. It is deprecated.

Update from the Nethermind: The issue was present in `Soulbound1155.sol`, which was removed and is not present in the final version.

6.14 [Best Practice] `__gap` variable

File(s): `upgradeables/ERCWhitelistSignatureUpgradeable.sol`, `upgradeables/ERCSoulboundUpgradeable.sol`, `upgradeables/AvatarBoundV1.sol`

Description: The `__gap` variable is used in upgradeable contracts that are inherited by other contracts. It is used to reserve empty space to avoid shifts in the storage layout during contract update that modifies the number of occupied storage slots. Below, we list the best practices in using the `__gap` that are not met in the current project:

- `__gap` should be used only for upgradeable contracts that are going to be inherited. Currently, the `AvatarBoundV1(...)` is not inherited by other contracts, but it also contains `__gap`;
- the size of the `__gap` array should be calculated so that the amount of storage used by a contract always adds up to the same number (usually 50 storage slots). E.g., if the four storage slots are occupied by state variables, the gap should be `__gap[46]`. This helps to maintain the contract updates when the number of reserved contract storage is a round, easy-to-remember number, like 50;
- `__gap` is usually defined as the last storage variable;

Recommendation(s): Consider applying best practices to the usage of `__gap`.

Status: Fixed

Update from the client: This issue has been fixed.

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 the documentation

Libertas Omnibus project provided a separate document that specified the desired features of the NFT tokens. Moreover, the code contained informative comments. The audit team remained in constant communication with the client during the audit.

8 Test Suite Evaluation

8.1 Hardhat Test Output

```
Compiled 110 Solidity files successfully (evm target: london).

  MockSoulbound
    _soulboundToken - ERC721 - must bound the token id properly
    _soulboundToken - ERC721 - must bound the token id properly - safeTransferFrom
    _soulbound - ERC1155 - must bound the token id properly
    _batchSoulbound - ERC1155 - must bound the token id properly
    burn - ERC1155 - must burn/sync tokens correctly

  GameSummary1155
    As Player must mint game summary achievement
    As Admin must mint a game summary for a player
    As Admin must mint a game summary but not the same game summary twice
    The pause functionality should works as expected
    As admin must mint a batch of game summaries achievement for players
    As game creator role I can create game summaries
    As an User I can get 1 GameSummary that the admin minted per game
    As an User I can get 1 GameSummary that the admin minted per game and check the PlayerData
    If the admin update one GameSummary, everyone will have this update
    As an Admin the BaseURI functionality should works
    As user I cant transfer/sell any achievement if is a SBT
    As user I can transfer/sell any achievement if is not a SBT
    As user I cant transfer any SBT using the safeBatchTransferFrom
    As a user I could burn any not SBT
    As a user I can burn any SBT but the info of the player will be cleared
    should revert if a non-admin tries to set a signer
    should revert if a non-admin tries to remove a signer
    should return true for supported interfaces
    should return false for unsupported interfaces
    should return the correct uri for a given token ID
    should allow batch burning of tokens
    should revert if the signature is not valid
    as admin should update the qty of achievements for a player
    as admin should update the qty of achievements for a player using the batch
    should update the qty of achievements for a player using the signature
    the batchUpdate using the signature should works as expected
    As admin or user minting shouldn't have collisions between ids

  LevelsBound
    As admin I can mint levels for a player
    As user I can't mint levels for a player
    The user can't have the same level token twice
    Sent the level 0 as new level is not allowed
    User only can lvl up once per level, more than once is not allowed
    Level down is not allowed
    Level up to the level 1 twice is not possible
    As user I can't transfer the level tokens
    As user I can't transfer the level tokens using the batch as well
    As user I can burn my level tokens
    LevelUp
      playerLevel should increase when mintLevel()
      playerLevel should reset to 0 when burn() or burnBatch()

  LevelsBoundV1
    As admin I can mint levels for a player
    As user I can't mint levels for a player
    The user can't have the same level token twice
    Sent the level 0 as new level is not allowed
    User only can lvl up once per level, more than once is not allowed
    Level down is not allowed
    Level up to the level 1 twice is not possible
    As user I can't transfer the level tokens
    As user I can't transfer the level tokens using the batch as well
    As user I can burn my level tokens
    LevelUp
      playerLevel should increase when mintLevel()
      playerLevel should reset to 0 when burn() or burnBatch()
```

```
Soulbound1155
    burn - ERC1155 - must burn tokens correctly
   Token Exists
      should fail to mint if putting wrong token id and vice versa
   Pause Mint
      should fail to mint if contract is paused and vice versa
   Verify Signature
      should fail when try to use invalid signature
      should fail when try to reuse used signature with mint()
      should fail when try to reuse used signature with mintBatch()
   Mint
      must bound the token id properly
      fail if try to mint more than the limit
      fail if already minted
      fail if try to mint invalid tokenId
      fail sender has no minter role
      adminMint should pass
   BatchMint
      must bound the token id properly
      fail if try to mint more than the limit
      fail if already minted
      fail if try to mint invalid tokenId
      fail sender has no minter role
      adminMintBatch should pass
   Token URI
      Get Uri() should fail if tokenId not exists
      Get Uri() should return tokenUri if tokenId exists
      updateBaseUri() should fail to update new baseuri if has no manager role
      updateBaseUri() should pass to update new baseuri if has manager role
   Token Transfer
      should able to transfer non-soulbound token
      should not able to transfer soulbound token
      should only transfer to/from whitelist address only
   Token Royalty
      should have default royalty on deploy
      Manager role should be able to update royalty

 83 passing (6s)
```

8.2 Foundry Test Output

```
forge test
[] Compiling...
[] Compiling 12 files with 0.8.17
[] Solc 0.8.17 finished in 116.16s

Running 7 tests for test/AvatarBound.t.sol:AvatarBoundTest
[PASS] testAdminMint() (gas: 331108)
[PASS] testFailUnauthorizedTransfer() (gas: 86792)
[PASS] testPauseUnpause() (gas: 296067)
[PASS] testSetBaseSkin() (gas: 60641)
[PASS] testSetBaseURI() (gas: 19950)
[PASS] testSetContractURI() (gas: 20419)
[PASS] testSetTokenURI() (gas: 272848)
Test result: ok. 7 passed; 0 failed; 0 skipped; finished in 6.98ms

Running 25 tests for test/Soulbound1155.t.sol:Soulbound1155Test
[PASS] testAdminMint() (gas: 115540)
[PASS] testAdminMintBatch() (gas: 213555)
[PASS] testAdminMintBatchNotMinterRole() (gas: 40202)
[PASS] testAdminMintNotMinterRole() (gas: 37658)
[PASS] testBurn() (gas: 290188)
[PASS] testBurnBatch() (gas: 584760)
[PASS] testBurnBatchNotOwnerShouldFail() (gas: 271476)
[PASS] testBurnNotOwnerShouldFail() (gas: 151871)
[PASS] testInvalidSignature() (gas: 37818)
[PASS] testMintAlreadyMinted() (gas: 196730)
[PASS] testMintInvalidTokenId() (gas: 60136)
[PASS] testMintMoreThanLimit() (gas: 88130)
[PASS] testMintShouldPass() (gas: 286749)
[PASS] testNonSoulboundTokenTransfer() (gas: 156514)
[PASS] testPauseUnpause() (gas: 135299)
[PASS] testReuseSignatureMint() (gas: 168388)
[PASS] testSoulboundTokenNotTransfer() (gas: 171902)
[PASS] testSoulboundTokenTransferOnlyWhitelistAddresses() (gas: 245496)
[PASS] testTokenExists() (gas: 206812)
[PASS] testTokenRoyaltyDefault() (gas: 13312)
[PASS] testTokenURIIfTokenIdExist() (gas: 37368)
[PASS] testTokenURIIfTokenIdNotExist() (gas: 13617)
[PASS] testUpdateTokenRoyalty() (gas: 23331)
[PASS] testUpdateTokenURIFailNotManagerRole() (gas: 38219)
[PASS] testUpdateTokenURIPass() (gas: 47040)
Test result: ok. 25 passed; 0 failed; 0 skipped; finished in 10.32ms

Running 34 tests for test/ItemBound.t.sol:ItemBoundTest
[PASS] testAddAlreadyExistingToken() (gas: 238472)
[PASS] testAddNewTokens() (gas: 610622)
[PASS] testAdminMint() (gas: 276461)
[PASS] testAdminMintId() (gas: 98477)
[PASS] testAdminMintIdNotMinterRole() (gas: 42600)
[PASS] testAdminMintNotMinterRole() (gas: 51024)
[PASS] testBurn() (gas: 550204)
[PASS] testBurnBatch() (gas: 569816)
[PASS] testBurnBatchNotOwnerShouldFail() (gas: 271459)
[PASS] testBurnIfHoldBothNonSoulboundAndSouldbound() (gas: 440744)
[PASS] testBurnNotOwnerShouldFail() (gas: 254656)
[PASS] testGetItemsPerTierPerLevel() (gas: 473391)
[PASS] testInvalidSignature() (gas: 51246)
[PASS] testMintInvalidTokenId() (gas: 61879)
[PASS] testMintMoreThanLimit() (gas: 73149)
[PASS] testMintShouldPass() (gas: 565499)
[PASS] testNonSoulboundTokenTransfer() (gas: 99427)
[PASS] testPauseUnpause() (gas: 114853)
[PASS] testPauseUnpauseSpecificToken() (gas: 383873)
[PASS] testReuseSignatureMint() (gas: 311813)
[PASS] testSoulboundTokenNotTransfer() (gas: 115316)
[PASS]
```

```
testSoulboundTokenTransferOnlyWhitelistAddresses() (gas: 198497)
[PASS] testTokenExists() (gas: 336892)
[PASS] testTokenRoyaltyDefault() (gas: 13459)
[PASS] testTokenURIIfTokenIdExistNOSpeficTokenURIFallbackToBaseURI() (gas: 120903)
[PASS] testTokenURIIfTokenIdExistWithSpeficTokenURI() (gas: 121587)
[PASS] testTokenURIIfTokenIdNotExist() (gas: 13816)
[PASS] testUpdateTokenBaseURIFailNotManagerRole() (gas: 38493)
[PASS] testUpdateTokenBaseURIPass() (gas: 148038)
[PASS] testUpdateTokenInfoCurrentMaxLevelShouldChange() (gas: 210264)
[PASS] testUpdateTokenRoyalty() (gas: 23396)
[PASS] testUpdateTokenURIFailNotManagerRole() (gas: 38785)
[PASS] testUpdateTokenURIPass() (gas: 148235)
[PASS] testgetAllItems() (gas: 134987306)
Test result: ok. 34 passed; 0 failed; 0 skipped; finished in 259.37ms

Ran 3 test suites: 66 tests passed, 0 failed, 0 skipped (66 total tests)
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

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

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

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

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

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

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

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

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

Learn more about us at nethermind.io.

General Advisory to Clients

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

Disclaimer

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