TECHFUND Audit Score

Smart contract Audit Report

BeyondConcept Audit

Completed on 2023-08-03

Score **POSITIVE+**

Risk level		
	Critical	0
	High	0
	Medium	0
	Low	2
	Note	2

Risk level detail

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

The tester arrives at the likelihood and impact estimates, they can now combine them to get a final severity rating for this risk. Note that if they have good business impact information, they should use that instead of the technical impact information.

https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

Vulnerability Review

Number of warnings

Compiler Version	1
Array optimisation for large arrays in case of (mint) Minter721	1
Coding convention for events emitted during add / update	1
Missing Checks Effects Interactions pattern	1
Checks for ERC721 finalisation	0
Using platform independent path construction to read artifacts	0
Prevent accessing same variable multiple times while getting extra headers	0
Error in setting up Dynamic Arrays	0
Gas wastage because of un-necessary bound checks	0
Re-Entrancy (out of order events)	0
Double Withdrawal	0

TECHFUND Audit Score

Smart contract Audit Report

Compiler Version

1

Several parts of contracts use not fixed compiler version. Different compiler versions can lead to different bytecode and gas costs for your contracts. Using a fixed compiler version helps you predict and control the behavior of your smart contracts on the Ethereum network.

It is highly recommended to use the same Solidity version throughout the codebase to maintain consistency and reduce the potential for version-specific vulnerabilities or compatibility issues.

```
pragma solidity ^0.8.18;

import './interfaces/IMinter721.sol';
import './interfaces/IReceiptStore721.sol';
import './interfaces/IWalletLink.sol';
import './interfaces/IMintableERC721.sol';

import '@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol';
import '@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol';
import '@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol';
```

Array optimisation for large arrays in case of (mint) Minter721

1

The mint function in Minter721 mint is used to mint multiple 721 tokens at once. It takes three arrays as input: _hashedUids, _tokens, and _tokenIds. A potential concern with this implementation is the use of external function calls, _verify and _mint, within a loop. External calls can be expensive in terms of gas costs, and executing them within a loop could lead to unexpected results.

```
for (uint256 i = 0; i < _hLen; i++) {
    _verify(_hashedUids[i], _tokens[i], _tokenIds[i]);
    _mint(_hashedUids[i], _tokens[i], _tokenIds[i]);
}
```

It's a good practice to set a limit on the maximum count of tokens that can be minted in a single transaction to prevent potential performance issues and excessive gas costs. Here's an updated version of your mint function that includes a parameter for specifying the maximum count of tokens to mint in a single transaction:

```
uint256 public maxTokensPerMint = 10;

function mint(
    string[] calldata _hashedUids,
    address[] calldata _tokens,
    uint256[] calldata _tokenIds
) external {
    require(_hashedUids.length <= maxTokensPerMint, "Exceeds max tokens per mint");
    uint256 _hLen = _hashedUids.length;
    if (_hLen != _tokens.length || _hLen != _tokenIds.length)
        revert LengthMismatch();

    for (uint256 i = 0; i < _hLen; i++) {
        _verify(_hashedUids[i], _tokens[i], _tokenIds[i]);
        _mint(_hashedUids[i], _tokens[i], _tokenIds[i]);
    }
}
```

Coding convention for events emitted during add / update

1

The setMinter function in the ReceiptStore contract is used to set the address of a new minter. The minter is an account that is authorized to mint or create new tokens (or receipts in this case) within the ReceiptStore contract. It's important to emit an event when changes like this occur in a smart contract, as emitting events provides a transparent and trackable way for external applications and users to monitor and react to changes happening on the blockchain.

The current implementation of the setMinter function lacks an event emission, which can make it difficult for external parties to detect and respond to changes in the minter address.

```
function setMinter(address _minter) external onlyAdmin {
    if (_minter == address(0)) revert AddressZero();
    minter = _minter;
 }
```

Similarly we also suggest adding events in DenonbuCheki

```
function setMetadataPatterns(uint _patterns) public onlyOwner {
    require(_patterns > METADATA_PATTERNS, "Invalid _patterns");
    METADATA_PATTERNS = _patterns;
}
```

```
function setMinter(address _minter) public onlyOwner {
    minter = _minter;
}
```

TECHFUND Audit Score

Smart contract Audit Report

Missing Checks Effects Interactions pattern

1

The "Checks, Effects, Interactions" (CEI) pattern is a best practice in Solidity smart contract development that helps ensure the security and proper functioning of your contracts. It consists of three main steps that should be followed in order:

Checks: Validate inputs and conditions before making any changes to the state or interacting with other contracts. This is to prevent invalid or malicious inputs from causing unexpected behavior.
Effects: Update the contract's internal state after performing necessary checks. This includes modifying variables, changing storage, and preparing data for interactions.
Interactions: Interact with external contracts, which typically involves making external function calls. This step is done after the internal state has been updated to ensure that interactions are based on the most accurate and up-to-date information.

Placing the event emitter ahead of the `interactions` ensures the log is updated and correct. It's "optimistic accounting". Should the `transfer` fail for some reason, it will cause a revert and there will be no event emitted. But in code , the event are emitted after interaction (mint).

```
function _mint(
    string calldata _hashedUid,
    address _token,
    uint256 _tokenId,
    bytes calldata _data
) internal {
    address _recipient = walletLink.getWalletAddress(_hashedUid);

    IMintableERC721(_token).mint(_recipient, _tokenId, _data);

    emit Mint(_hashedUid, _token, _tokenId);
}
```