# 660.https://stackoverflow.com/questions/67611716/why-does-the-minting-function-of-erc721-have-an-access-control

**T:**Why does the minting function of ERC721 have an access control?

**Q:**Most of the ERC721 examples using Open Zeppelin I see require the mint function to have an access control where only the owner of the contract is allowed to call the function. For example,  
  
function mint(address to) public virtual { require(hasRole(MINTER\_ROLE, \_msgSender()), "ERC721PresetMinterPauserAutoId: must have minter role to mint"); \_mint(to, \_tokenIdTracker.current()); \_tokenIdTracker.increment();}  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
or the following using the Ownable library.  
  
function mint(address receiver) external onlyOwner returns (uint256) { \_tokenIds.increment(); uint256 newTokenId = \_tokenIds.current(); \_mint(receiver, newTokenId); return newTokenId;}  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Does this mean a new contract has to be deployed each time a new token is minted? This seems not only excessive in terms of the gas fee, but also the ERC721 contract has properties for mapping different owners and tokens:  
  
// Mapping from token ID to owner addressmapping (uint256 => address) private \_owners;// Mapping owner address to token countmapping (address => uint256) private \_balances;  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
which wouldn't make sense if minting is restricted to the contract owner.  
  
It makes more sense to me that you deploy a single ERC721 contract (and its dependencies) and have the users call the mint function. What is the best practice for the mint function of ERC721?

1 **Answer**

**A1:**The ERC-721 standard does not define a "best" or "correct" way to mint new tokens (such as whether it should be open or restricted) and it's up to each contract developer to implement or omit the minting feature in a way that reflects their needs.  
  
Creating of NFTs ("minting") and destruction NFTs ("burning") is not included in the specification. Your contract may implement these by other means. Please see the event documentation for your responsibilities when creating or destroying NFTs.  
  
But having a whitelist of addresses that are authorized to mint new tokens (e.g. MINTER\_ROLE or onlyOwner) seems to be more common than allowing anyone to freely mint new tokens.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Even though it's theoretically possible to deploy new contract each time a new token is minted, it's not a standard approach (and I personally haven't seen any contract that does it). In most cases the minting process "just" creates a new ID, stores a new string/URL value associated with the ID, associates this new token with an owner address (of the token, not a contract owner), plus updates some metadata such as amount of tokens owned by an address (see example below).  
  
The token owner can then transfer their tokens, give anyone control over their tokens, and do other stuff depending on the contract implementation.  
  
The mappings that you point out in your question (\_owners and \_balances) suggest that they store token owner (not contract owner) addresses as well as amount of tokens held by each address.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Example:  
  
 ● Contract owner mints token ID 1 to address 0x123.  
  
Value of \_owners[1] is 0x123 (was 0, the default value)  
  
Value of \_balances[0x123] becomes 1 (was 0, the default value)  
  
  
  
  
 ● Contract owner mints token ID 2 to address 0x123.  
  
Value of \_owners[1] is still 0x123  
  
Value of \_owners[2] is now 0x123 (was 0, the default value)  
  
Value of \_balances[0x123] becomes 2 (because they now own 2 tokens)

**C1:**Are the ERC721 contracts mentioned here only useful if the contract owner is minting tokens for certain addresses? I'm actually looking for a structure where users can mint their own tokens without the central figure. Is there any security flaws if I were to simply remove the access control for the mint function and make it public? I'm just trying to make sense of the reason for the access limitation because I'm not sure if it's as important as the access control for transferring or burning a token.

**C2:**If you want to open the minting feature, you can remove the authorization from the mint() function. From security standpoint, I can only think of higher probability of reaching the ID max value (if you chose to make it uint8 or some "small" datatype) and possible integer overflow (if you don't prevent it either using Solidity 0.8+, or checking with require/assert). Otherwise, it's same level of security as if you only had authorized addresses, because they should be treated in the code as untrusted as well (e.g. don't trust but verify that they return some value if they're a contract).

**C3:**Is it a more common approach to deploy a new contract when the URL is different (to signify a different creature as opposed to just a different variant) or is it better to re-use the old contract and keep adding different URLs for different things to the same contract?

**C4:**@KernelJames A general practice is one contract for one collection. So if you have a collection of "cat NFTs", you might want to deploy a new contract for another collection of "dog NFTs". But if your overal aim is to create a collection of "animal NFTs", they you might want to group all these tokens under the one collection.