J Jeffrey Scholz Nov 8, 2023 9 min read

A comprehensive guide to the ERC 721 standard and related security issues

Updated: 3 days ago

ERC721 (or ERC-721) is the most widely used Ethereum standard for nonfungible tokens. It associates a unique number with an Ethereum address, thereby denoting that address owns the unique number -- the NFT.

There is indeed no shortage of tutorials covering this famous token design, however, we have found that many developers, even experienced ones, do not have a complete understanding of the specifications — and sometimes the security issues. Therefore, we documented the standard here with an emphasis on areas more experienced developers miss.

Practice problems are provided at the end to test the lesser known corner cases.

Table of Contents

What makes NFTs unique?

NFTs are uniquely identified by the three values (chain id, contract address, id).

Owning an NFT means owning a uint256 stored in an ERC721 contract on a particular EVM chain.

We will delve into the functions that make up the ERC721 spec and facilitate its behavior, including core and auxiliary functions. They are:

- **ownerOf**: Ownership Mapping
- **mint**: Token Creation
- **transferFrom**: Transferring Ownership
- **balanceOf**: Ownership Count
- **setApprovalForAll** & **isApprovedForAll**: Delegating Transfer Rights
- **approve** & **getApproved**: Single NFT Approval Mechanism
- **safeTransferFrom** & **_safeMint**: Secure Transfer Functions
- **burn**: NFT Destruction

Ownership and the ERC721 **ownerOf** function

Ownership is just a mapping: ownerOf(uint256 id)
At it's core, an ERC721 is just a mapping from a uint256 (the id of the NFT) to the address of the owner. For all the hype about NFTs, they are glorified hash maps. "Owning" an NFT means there is a mapping that has a certain id as the key and your address as the value. That's all.

For the sake of simplicity, we will use a public variable instead of a public function. On the outside, the interaction is identical.

The function (or public mapping) ownerOf takes the id of the NFT and returns the address that owns it.

Minting process with **mint** function

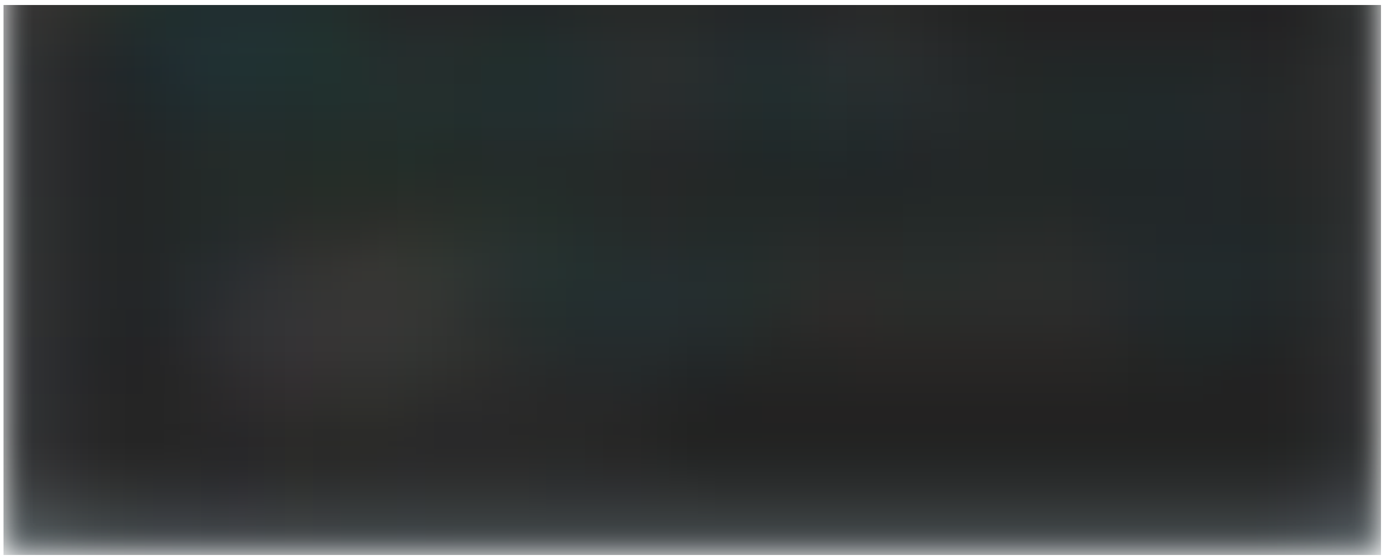
Since the default value of a mapping is 0, by default, the zero address "owns" all the NFTs, but this is not how we generally interpret this. If ownerOf returns the zero address, we say the NFT does not exist. Minting is how tokens come into existence.

Mint is not part of the ERC721 spec, it is left up to the user to define how NFTs get minted. There is no requirement that NFTs be minted in the sequence 0,1,2,3, etc. We could mint someone an NFT based on the block number hashed with their address or something like that. In the following implementation, anyone can mint any id as long as it hasn't been minted before.

It may seem funny to have a **Transfer** event going from **address(0)** to the recipient, but that's the spec.

Transferring NFTs with ERC721 **transferFrom**

Naturally, we want a way to move our NFT to another address. The function transferFrom accomplishes this.



It may seem odd that transferFrom is payable, but that is what the EIP 721 spec says. Presumably, this is to allow for applications that require payment of Ether for acquiring an NFT that has already been minted. Many implementations do not follow this part of the specification, and this feature is very rarely used.

Also, why do we have a from field if we only allow msg.sender to be the from? We will get to this when we talk about approvals. For now, it should be obvious that the owner should be able to transfer an id they own.

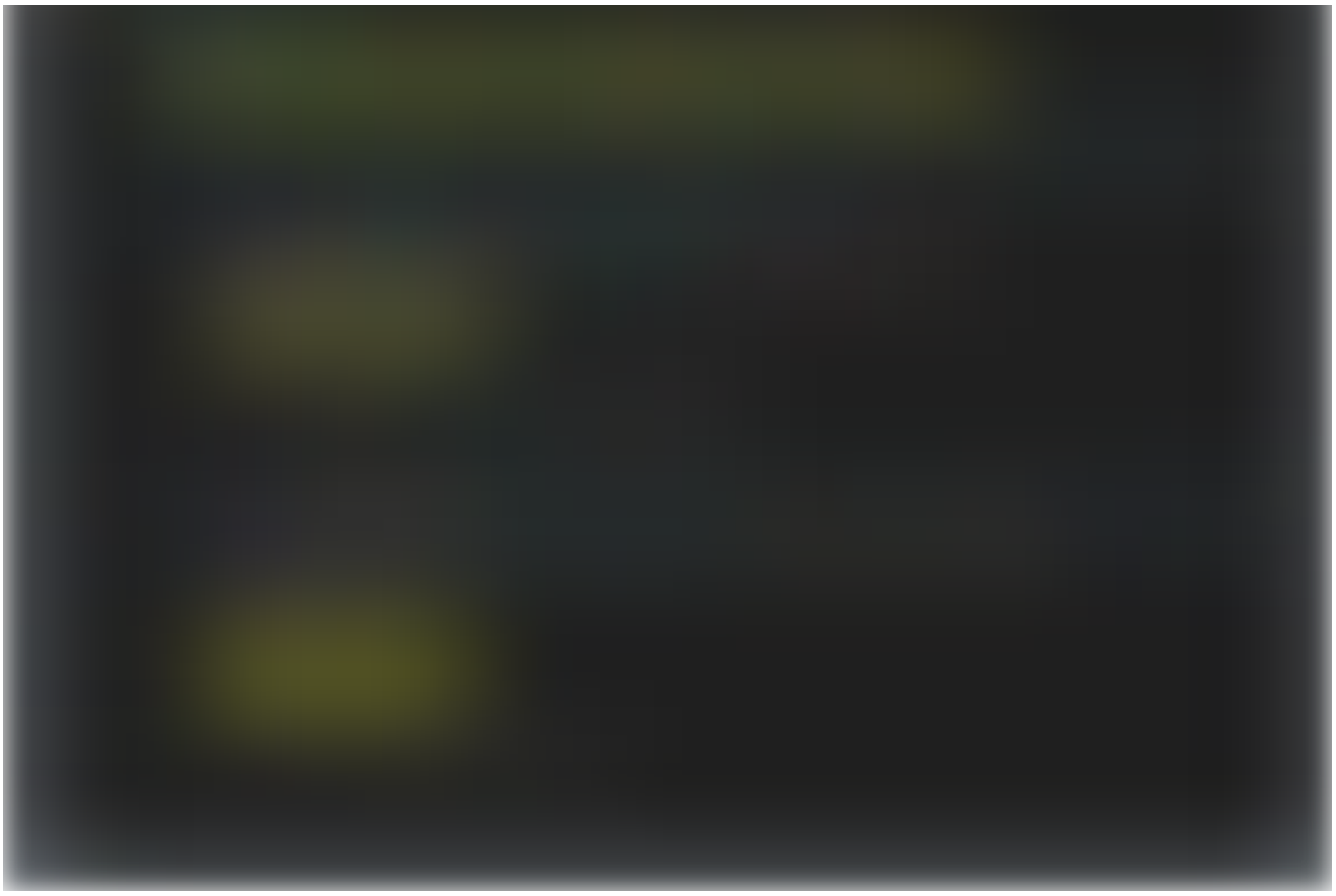
Understanding the ERC721 **balanceOf** function

The ERC721 specification requires us to track how many NFTs an address owns per contract.

ERC721 holds a mapping **mapping(address owner => uint256 balances)** balanceOf.

Our minimal NFT now has the following functionality shown in the code below.

It should be emphasized that **balanceOf** only says how many NFTs an address owns, it doesn't say which ones. We need to update the functions where the balances can change, which of course are **mint** and **transfer**. The places we updated those functions have been highlighted

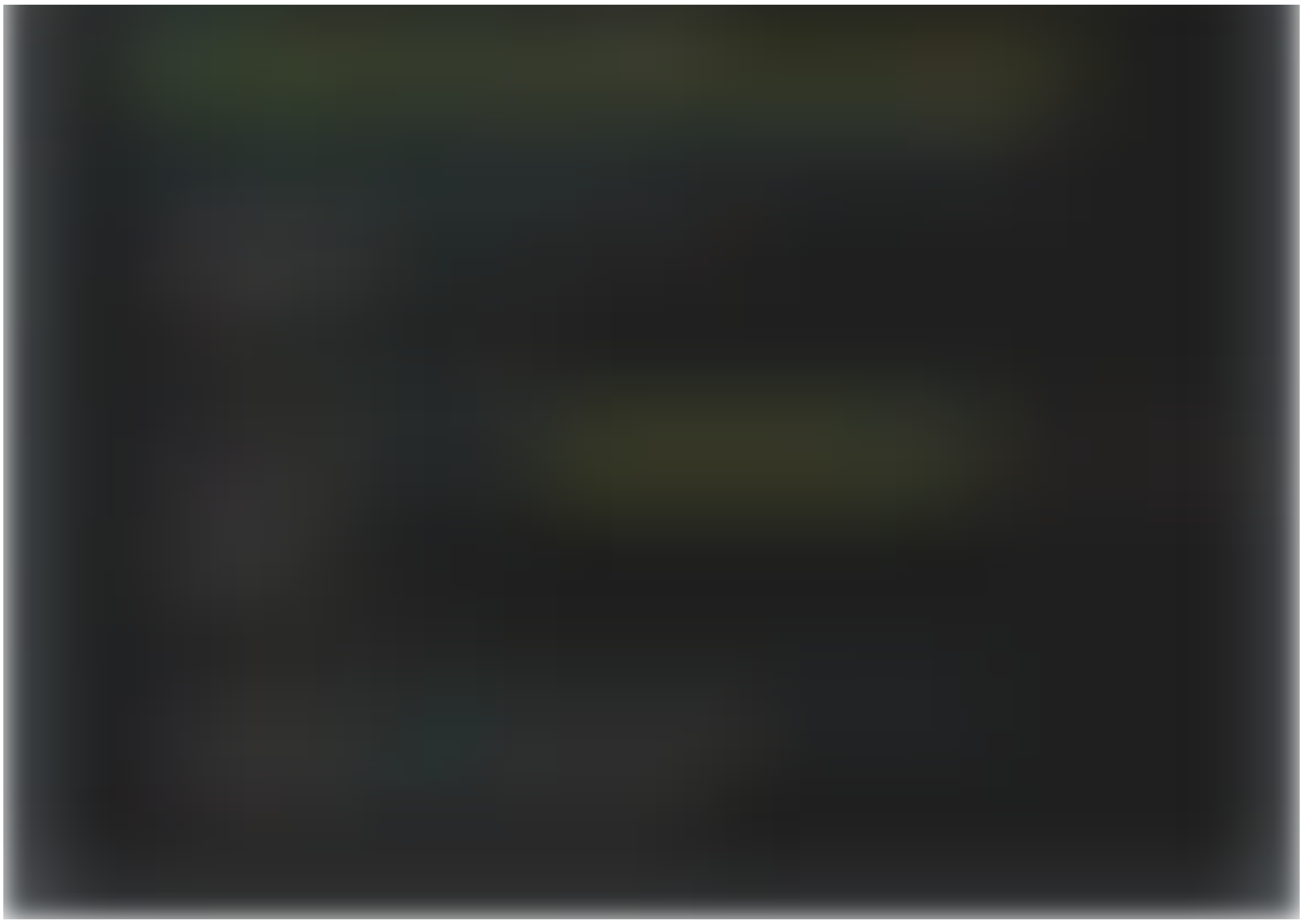


Here is another warning: **an owner can transfer NFTs at will, so you should be very careful when relying on balanceOf when making decisions in a smart contract. Do not treat balanceOf() like a static value as it can change over the course of the transaction if the owner transfers an NFT to himself from another address, or transfers the NFT to another address they own, they can manipulate the balanceOf() function.**

Unlimited approvals: ERC721 **setApprovalForAll** and **isApprovedForAll** functions

The ERC721 specification allows an NFT owner to give control of the NFT to another address without transferring the NFT to them. The first mechanism to do this is with the **setApprovalForAll()** function. As the name implies, it allows another address to transfer NFTs on behalf of the owner. **This applies to any NFT the address owns.** The counterpart **isApprovedForAll()** checks if a certain address called the operator has been delegated authority from an owner.

An owner can have multiple operators. This is a mechanism by which the same NFT can be for sale on multiple NFT marketplaces. If the market places are approved for the owner's address, they can transfer it to a buyer if the buyer pays the right amount of Ether.



TransferFrom now allows both the owner and an address which has is **_approvedForAll** to transfer the token.

Specific token id approvals with ERC721 **approve** and **getApproved** functions

Instead of approving another address to be able to transfer every NFT you own, you can approve them for a single id, which is generally safer. This is put into the public mapping **getApproved()**.

Unlike **isApprovedForAll**, being approved for an NFT has nothing to do with the owner's address, it is only associated with the id.

After a transfer, the new owner probably doesn't want someone else to have approval over that id. Therefore, the transferFrom function needs to be updated to clear that approval.

A limitation of **approve** is that only one address can be approved per **id**. If we want to approve multiple addresses, it would be very expensive to delete all of them during a transfer.

Note that if an address is **approvedForAll**, then it is able to **approve** another address for ids owned by the address it is an operator for. Nothing has changed in the setApprovalForAll() function.

After the transfer, approvals are cleared because the new owner in general will not want the preview address to have approval over the id.

We have nearly completed implementing every function the ERC721 specification requires. The remaining ones require significantly more documentation.

Identifying owned NFTs without the enumerable extension

Determining a list of ids owned

Using the methods above, is there an efficient way to determine which NFTs an address owns?

There isn't.

The function balanceOf only tells us how many NFTs an address owns, and ownerOf only tells us who owns a particular id. In theory, we could loop through all the ids to figure out which ones a particular address owns, but this is not efficient.

If a contract needs to know that 0xc0ffee… owns ids 5, 7, and 21, the solution is to ***tell*** the contract 0xc0ffee… owns those ids, then the contract verifies it is in fact true.

But how do we efficiently determine 0xc0ffee… owns 5, 7, and 21 off chain? We could loop through all the ids and call ownerOf(), but that would make our RPC provider rich.

Parsing ERC721 Events

Here is some sample code using web3 js to track which NFTs are owned by an address. Be aware the code scans events since the 0th block, which is not efficient. You should pick a more sensible recent value.

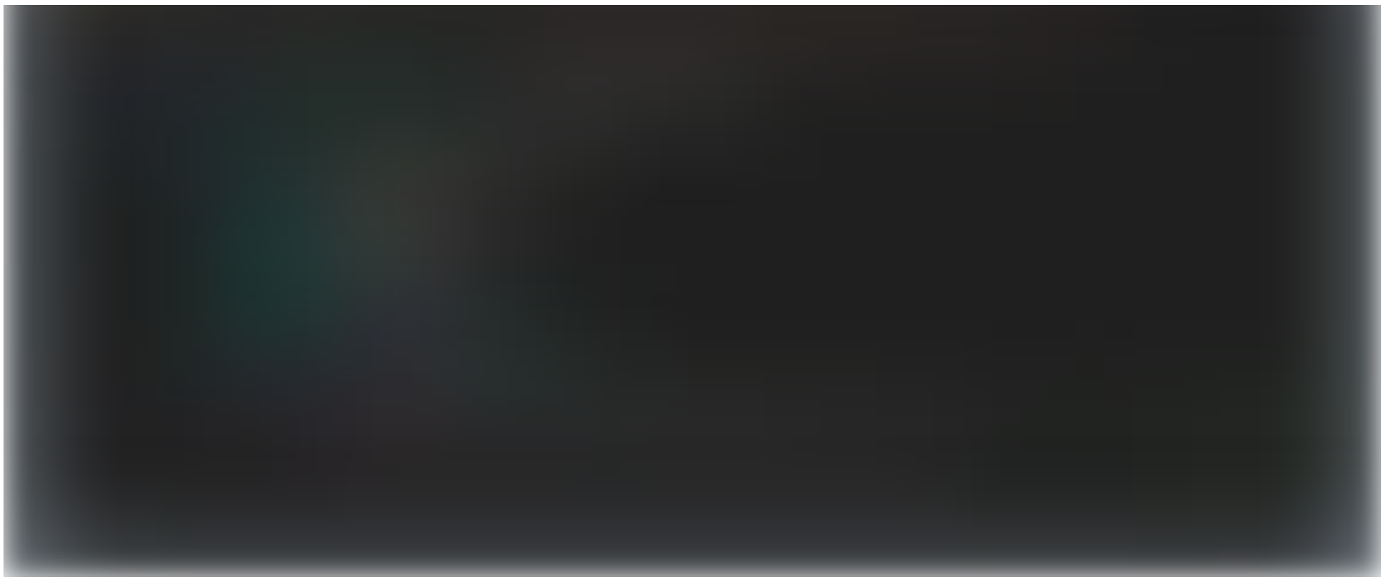
gist.github.com/RareSkills/5d60ad42cdd81b6e136605a832ba59ee

Safe Transfers: **safeTransferFrom** , **_safeMint** and the **onERC721Received** function

The intent of **safeTransferFrom** and **_safeMint** is to handle NFTs getting stuck in a contract. If an NFT is transferred to a contract that does not have the ability to call transferFrom itself, then the NFT will be "locked in" the contract, effectively destroying it.

To prevent this from happening, ERC721 only wants to transfer to contracts that have a mechanism to be able to transfer away the NFT at a later time. A contract is marked as being able to "handle" NFTs if it has a function **onERC721Received()** which returns the magic bytes4 value 0x150b7a02. This is the function selector of **onERC721Received()** show below. (A function selector is Solidity's internal identifier for functions).

Here is a minimal example of a contract using that interface:



safeTransferFrom behaves exactly like **transferFrom**. Under the hood it calls transferFrom **and then** checks if the receiving address is a smart contract.

- If it is not, don't do any additional steps
- If it is
  - It tries to call the function **onERC721Received()** with the arguments above on the contract receiving the NFT
  - If the function call reverts or 0x150b7a02 is not returned, it reverts

Why check for the function selector?

Checking if **onERC721Received()** didn't revert isn't good enough to determine if a contract can properly handle ERC721 tokens.

If an NFT is transferred to a smart contract with a fallback function, and the return value is not checked for, the transaction will not revert. However, the contract probably does not have a mechanism to handle receiving NFTs just because it has a fallback function.

Function arguments of onERC721Received

When **onERC721Received** is called, the following arguments are passed to it, which are described below

operator:
Operator is **msg.sender** from the perspective of safeTransfer. It could be the NFT owner or an address approved to transfer that NFT.

from:
From is the owner of the NFT. The parameters from and operator will be equal if the owner is the one calling transfer.

tokenId:
The **id** of the NFT being transferred.

data:
If **safeTransferFrom** was called with **data**, this is forwarded to the receiving contract. The **data** parameter will be discussed in a later section.

it doesn't have. If your contract uses **onERC721Received()**, you must check that msg.sender is the NFT contract you expect!

safeTransfer reentrancy
SafeTransfer and _safeMint hand execution control over to an external contract. **Be careful when using safeTransfer to send an NFT to an arbitrary address, the receiver can put any logic they like the onERC721Received() function, possibly leading to reentrancy.** If you properly defend against reentrancy, this does not need to be a concern.

safeTransfer denial of service
A malicious receiver can forcibly revert transactions by reverting inside onERC721Received() or by using a loop to consume all the gas. You should not assume that safeTransferFrom to an arbitrary address will succeed.

safeTransferFrom with data and why it exists - Practical Use Cases and Efficiency

ERC721 specifies that two safeTransferFrom functions exist:

[illegible]

The second one has an additional **data** parameter. The following example will demonstrate using the data parameter with **onERC721Received()**.

Gas efficient staking, bypassing approval

A very common pattern is to deposit an NFT into a contract for the purpose of staking. Of course the NFT is not "inside" the smart contract, but rather the **ownerOf** that particular id is the staking contract, and the staking contract has some book keeping to keep track of the original owner.

A common, but inefficient way to do it is shown in the following code snippet. The reason it is inefficient is because it requires the user to **approve** Staking before they call **deposit()**. We've added the option to vote while staking as an example of adding parameters during a transfer.

Recent Posts

See All

Understanding the Function Selector in Solidity

18 0 1

sset i

rrors

How ERC721 Enumerable Works

178 0 2

argument.

Subscribe to our newsletter

Subscribe Now

We do not sell your information to anyone. Period.

The **burn** function and NFT Destruction

An NFT can be burned by transferring it to the zero address. Being able to burn NFTs is not officially part of the ERC specification, so contracts are not required to support this operation.

ERC721 Implementations

The OpenZeppelin implementation is the most beginner friendly library for developers and is ideal if used with the rest of the upgradeable contracts. More experienced developers should consider the Solady ERC721 implementation as it will offer considerable gas savings.

Test your knowledge

Because ERC721 is so ubiquitous, serious Solidity developers should understand the protocol entirely and be able to implement one from scratch by memory. To see if you understood everything, try solving the following security exercises for ERC721:

Overmint 1 (RareSkills Riddles)
Overmint 2 (RareSkills Riddles)
Diamond Hands (RareSkills Riddles)
Jpeg Sniper (Mr Steal Yo Crypto)

Keep learning: ERC721 Enumerable

The Enumerable extension to ERC721 allows a smart contract to list all the NFTs owned by an address. See our our article about ERC721 Enumerable to continue learning.

Learn More with RareSkills

Please see our industry leading Solidity bootcamp to learn more about the program.