Guide to Ownership and Access Control in Solidity

From the simple to the complex, with the code to reuse.

Alberto Cuesta Cañada

Follow

Coinmonks

--

1

Listen

Share

May 2020 update:

Smart contract development moves really fast, and the situation described in this article has changed. I collaborated with OpenZeppelin to develop the access control libraries in their 3.0 release and of the smart contracts described here, only Ownable.sol is still in use. Please, readthis other article for an updated guide on using access control in your smart contracts.

Introduction

When writing smart contracts I tend to take an educational approach. Even if they are intended for a production environment I make them as easy to understand as possible. I write contracts to be reusable, but usually they get rewritten for each specific business case.

In this article I'm going to discuss three approaches to permissioning in solidity smart contracts. These approaches are discussed in increasing order of complexity, which is the order in which you should consider them for your project. I include code that you can reuse for each approach.

This article assumes that you are comfortable coding smart contracts in solidity, and using features like inheritance and passing contracts addresses as a parameter. If you are looking for an easier article on smart contract development you can try this one.

New to Crypto? Try crypto trading bots or copy trading

Simple Approach — Ownable.sol

The Ownable.sol contract from OpenZeppelin must be one of the most reused contracts out there. In 77 lines it implements:

1. The logic to assert that someone is the owner of a contract.
2. The logic to restrict function calling to the contract owner for inheriting contracts.
3. The logic to transfer ownership to a different address.

When coding smart contracts you will inherit from Ownable

very often. Let's see how to use Ownable

with an example. Imagine that you want to keep a list of addresses in a contract but you want to be the only one that can add more. Think of it like some kind of registry of people that you trust. You could do something like:

Inheriting from Ownable

and calling its constructor on yours ensures that the address deploying your contract is registered as the owner. The onlyOwner

modifier makes a function revert if not called by the address registered as the owner.

Once you deploy this contract only you or someone that you designate can add new members to the list within.

That's it, in a nutshell. There are a couple more functions but you'll get it if you check thesource code. Try to implement

something simple with it, it is very easy to understand.

Despite its usefulness, there will be many times when Ownable

is not enough. Only one address can be the owner at a given time, only the owner gets to decide who can be the new owner, you can only check if you are the owner, not is someone else is.

Middle Approach — Whitelist.sol

Whitelist.sol keeps a list of addresses which then can be used to restrict functionality or any other purpose. It is very similar in functionality to OpenZeppelin's Roles.sol, although with some critical differences (check ourREADME).

Whitelist.sol only has three functions:

With this contract you could, for example, keep a list of approved stakeholders who can be the only recipients for token transfers. You could do something like this:

In the example above, you could also make ERC20Whitelisted

inherit from both ERC20

and Whitelist

. There are some trade offs thatI would be happy to discuss.

Simple whitelists can be quite powerful. OpenZeppelin implemented many ERC20 and ERC721 variants using them and managed to provide more functionality than most of us will need. At TechHQ we implemented CementDAO using only whitelists as well.

Sometimes, however, whitelists will also fall short. You might need to have more than one owner for a whitelist. Or you might need to manage many overlapping whitelists. For those cases we have a hierarchical role contract.

Complex — RBAC.sol

We developed RBAC.sol aiming to give multi user functionality like you have in modern shared systems.

1. There are roles that are nothing more than groups of addresses.
2. Group membership can only be modified by members of some administrator role.
3. New roles can be created at runtime.
4. Role membership can be verified.

At a low level we identify the roles using a bytes32

argument chosen by the user. Commonly these are identifiable short strings, but you can also use an encrypted value or an address.

The roles themselves are a group of member addresses and the identifier of the admin role. Funnily enough we don't need to store the identifier of the role inside its own struct.

There are now two methods to add a new role and verify if a role exists:

And the functions for managing members are the same, only that now the relevant role must be specified:

addMember

and removeMember

will only succeed if the caller belongs to the administrator role of the role that we are adding members to.

addRole

will only succeed if the caller belongs to the role that will administer the role being created.

These simple rules will allow to create a hierarchy of roles, which can then be used to implement complex multi user platforms with different permissioning levels or areas.

Further Learning

To dive even deeper into the rabbit hole I suggest starting with this issue from OpenZeppelin. Their codebase and ours is not that different, and you will find there a thorough reasoning for most design decisions even in the cases where we have chosen to go the other way. Their use of Roles

for contracts like ERC20Mintable

is a great example of an alternative to Whitelist

.

Another resource for the brave is the AragonOS ACL contract. Just a glance to the interface shows that they have decided to go farther than anyone else:

We use the three levels of access control described in this article for the examples in our own @hq20/contracts package, so you should also keep an eye there as well.

Conclusion

When it comes to smart contract implementation it is a good idea to implement only the complexity that is required, and no more. In terms of permissioning there are three distinct levels of complexity:

1. Single user
2. Group of users
3. Hierarchy of user groups

You can use Ownable.sol for systems that are permissioned for a single user. You can use @openzeppelin/Roles.sol or @hq20/Whitelist.sol for systems that require permissioning users in a group. For systems that require a group hierarchy we have successfully used @hq20/RBAC.sol in the past.

You will have your own requirements and will need to take your own decisions on trade offs. Knowing the design decisions behind each implementation will allow you to either use an existing contract or to modify one for your own use.

Please make sure of letting us know of any feedback. We are developing the @hq20/contracts package to support the coding of real world blockchain applications. We are aiming for our code to be reused and abused, and would be very happy to know how you do that.