Logging in with NEAR

Previously…

In the previous chapter we simply displayed whether the crossword puzzle was solved or not, by checking the solution hash against the user's answers.

Updates to transfer prize money

In this chapter, our smart contract will send 5 Ⓝ to the first person who solves the puzzle. For this, we're going to require the user to have a NEAR account and log in.

Better onboarding to come Later in this tutorial we won't require the user to have a NEAR account.

Since logging in is important for many decentralized apps, we'll show how this is done in NEAR and how it's incredibly unique compared to other blockchains. This transfer will occur when the first user to solve the puzzle calls thesubmit_solution method with the solution. During the execution of that function it will check that the user submitted the correct answer, then transfer the prize.

We'll be able to see this transfer (and other steps that occurred) inNearBlocks Explorer .

But first let's talk about one of the most interesting superpowers of NEAR: access keys.

Access keys

You might be familiar with other blockchains where your account name is a long string of numbers and letters. NEAR has an account system where your name is human-readable, likefriend.testnet for testnet orfriend.near for mainnet.

You can add (and remove) keys to your account on NEAR. There are two types of keys: full and function-call access keys.

The illustration below shows a keychain with a full-access key (the large, gold one) and two function-call access keys.

Art byalcantara_gabriel.near

Full-access keys

Full-access keys are the ones you want to protect the most. They can transfer all the funds from your account, delete the account, or perform any of the otherActions on NEAR .

When we used thenear login command in theprevious chapter , that command asked the full-access key in the NEAR Wallet to use theAddKey Action to create another full-access key: the one we created locally on our computer. NEAR CLI uses that new key to deploy, make function calls, etc.

Function-call access keys

Function-call access keys are sometimes called "limited access keys" because they aren't as powerful as the full access keys.

A Function-call access key will specify:

- What contract it's allowed to call
- What method name(s) it's allowed to call (you can also specify all functions)
- How much allowance it's allowed to use on these function calls

It's only allowed to perform theFunctionCall Action.

Example account with keys

Let's look at this testnet account that has one full-access key and two function-call access keys. As you can see, we use the NEAR CLIcommandkeys to print this info.

Let's look deeper into each key.

First key

{ access_key :

{ nonce :

72772126000000 ,

// Large nonce, huh! permission :

{ FunctionCall :

{ allowance :

'777000000000000000000000000' ,

// Equivalent to 777 NEAR method_names :

[ ] ,

// Any methods can be called receiver_id :

'puzzle.testnet'

// This key can only call methods on puzzle.testnet } } } , public_key :

'ed25519:9Hhm77W4KCFzFgK55sZgEMesYRaL8wV1kpqh8qntnSPV' } The first key in the image above is a function-call access key that can call the smart contractpuzzle.testnet onany method . If you don't specify which methods it's allowed to call, it is allowed to call them all. Note the empty array ([]) next tomethod_names , which indicates this.

We won't discuss the nonce too much, but know that in order to prevent the possibility of replay attacks , the nonce for a newly-created key is large and includes info on the block height as well as a random number.

The allowance is the amount, in yoctoNEAR, that this key is allowed to use during function calls. Thiscannot be used to transfer NEAR. It can only be used in gas for function calls.

The allowance on this key is intentionally large for demonstration purposes.777000000000000000000000000 yoctoNEAR is777 NEAR, which is unreasonably high for an access key. So high, in fact, that it exceeded the amount of NEAR on the contract itself when created. This shows that you can create an access key that exceeds the account balance, and that it doesn't subtract the allowance at the time of creation.

So the key is simply allowed to use the allowance in NEAR on gas, deducting from the account for each function call.

Second key

{ access_key :

{ nonce :

72777733000000 , permission :

{ FunctionCall :

{ allowance :

'250000000000000000000000' ,

// 0.25 NEAR, which is a typical allowance method_names :

[

'foo' ,

'bar'

] ,

// Can call methods foo and bar only receiver_id :

'puzzle.testnet' } } } , public_key :

'ed25519:CM4JtNo2sL3qPjWFn4MwusMQoZbHUSWaPGCCMrudZdDU' } , This second key specifies which methods can be called, and has a lower allowance.

Note that the allowance for this key (a quarter of a NEAR) is the default allowance when a person "logs in" in with the NEAR Wallet.

In NEAR, "logging in" typically means adding a key like this to your account. We'll cover this more in a moment.

Third key

{ access_key :

{

nonce :

72770704000019 ,

permission :

'FullAccess'

} , public_key :

'ed25519:FG4HjEPsvP5beScC3hkTLztQH8k9Qz9maTaumvPDa5t3' } The third key is a full-access key.

Since this key can perform all the Actions, there aren't additional details or restrictions like the function-call access keys we saw.

What does "log in" mean in a blockchain context?

Let's take a step back from NEAR and talk about how login works broadly using web3 wallets.

A web3 wallet (like Ethereum's MetaMask, Cosmos's Keplr, or the NEAR Wallet) stores a private key for an account. When interacting with decentralized apps, a user will typically use the wallet to sign transactions and send them to the blockchain for processing.

However, web3 wallets can also be used to sign any kind of message, and it doesn't need to send anything to the blockchain. This is sometimes called "offline signing" and protocols will sometimes create standards around how to sign data.

In other ecosystems, the idea of "logging in" with a web3 wallet uses this offline signing. A user is asked to sign a structured message and a backend can confirm that the message was signed by a given account.

NEAR keys can also sign and verify messages in this manner. In fact, there are a couple simple examples of how to achieve this in thenear-api-js cookbook .

There are potential drawbacks to this offline signing technique, particularly if a signed message gets intercepted by a malicious party. They might be able to send this signature to a backend and log in on your behalf. Because this all takes place offline, there's no mechanism on-chain to revoke your login or otherwise control access. We quickly see that using a web3 wallet for signed typed data runs into limitations.

So signing a message is fine, but what if we could do better?

With NEAR, we can leverage access keys to improve a user's login experience and give the power back to the user.

If I log into theGuest Book example site , I create a unique key just for that dApp, adding it to my account. When I'm done I can remove the key myself. If I suspect someone has control of my key (if a laptop is stolen, for example) I can remove the key as long as I have a full-access key in my control.

Logging in with NEAR truly gives the end user control of their account and how they interact with dApps, and does so on the protocol level.

The concept of access keys is so important that we've spent longer than usual on the topic without actually implementing code for our improved crossword puzzle.

Let's move to the next section and actually add the login button.Edit this page Last updatedonFeb 8, 2024 bymatiasbenary Was this page helpful? Yes No

Previous Add a puzzle Next Access keys and login 2/2