

Authenticate NEAR Users

Recently NEAR has approved a new standard that, among other things, enables users to authenticate into a backend service.

The basic idea is that the user will sign a challenge with their NEAR wallet, and the backend will verify the signature. If the signature is valid, then the user is authenticated.

Backend Auth with a NEAR Wallet

Authenticating users is a common use-case for backends and web applications. This enables services to provide a personalized experience to users, and to protect sensitive data.

To authenticate a user, the backend must verify that the user is who they say they are. To do so, the backend must verify that the user has access to a full-access key that is associated with their account.

For this three basic steps are needed:

1. Create a challenge for the user to sign.
2. Ask the user to sign the challenge with the wallet.
3. Verify the signature corresponds to the user.

1. Create a Challenge

Assume we want to login the user into our application named `application-name`.

We first need to create a challenge that the user will sign with their wallet. For this, it is recommended to use a cryptographically secure random number generator to create the challenge.

```
import
```

```
{ randomBytes }
```

```
from
```

```
'crypto' const challenge =
```

```
randomBytes ( 32 ) const message =
```

```
'Login with NEAR' note Here we use crypto.randomBytes to generate a 32 byte random buffer.
```

2. Ask the User to Sign the Challenge

The `signMessage` method needed to sign the challenge is supported by these wallets:

- Meteor Wallet
- Here Wallet
- Near Snap
- Nightly Wallet
- WELLDONE Wallet
- NearMobileWallet
- MyNearWallet
- Sender

The message that the user needs to sign contains 4 fields:

- Message: The message that the user is signing.
- Recipient: The recipient of the message.
- Nonce: The challenge that the user is signing.
- Callback URL: The URL that the wallet will call with the signature.

```
// Assuming you setup a wallet selector so far const signature = wallet . signMessage ( { message , recipient ,
```

```
nonce : challenge ,
```

```
callbackUrl :
```

```
< server - auth - url
```

```
})
```

3. Verify the Signature

Once the user has signed the challenge, the wallet will call the callbackUrl with the signature. The backend can then verify the signature.

```
const naj =
require ( 'near-api-js' ) const js_sha256 =
require ( "js-sha256" )

export
async
function
authenticate ( { accountId , publicKey , signature } )

{ // A user is correctly authenticated if: // - The key used to sign belongs to the user and is a Full Access Key // - The object
signed contains the right message and domain const full_key_of_user =

await

verifyFullKeyBelongsToUser ( { accountId , publicKey } ) const valid_signature =

verifySignature ( { publicKey , signature } ) return valid_signature && full_key_of_user }

export
function
verifySignature ( { publicKey , signature } )

{ // Reconstruct the payload that wasactually signed const payload =

new
Payload ( {

message :

MESSAGE ,

nonce :

CHALLENGE ,

recipient :

APP ,

callbackUrl : cURL } ) ; const borsh_payload = borsh . serialize ( payloadSchema , payload ) ; const to_sign =

Uint8Array . from ( js_sha256 . sha256 . array ( borsh_payload ) )

// Reconstruct the signature from the parameter given in the URL let real_signature =

Buffer . from ( signature ,

'base64' )

// Use the public Key to verify that the private-counterpart signed the message const myPK = naj . utils . PublicKey . from (
publicKey ) return myPK . verify ( to_sign , real_signature ) }

export
async
function
verifyFullKeyBelongsToUser ( { publicKey , accountId } )

{ // Call the public RPC asking for all the users' keys let data =
```

```

await
fetch_all_user_keys ( { accountId } )

// if there are no keys, then the user could not sign it! if
( ! data ||
! data . result
||
! data . result . keys )
return
false

// check all the keys to see if we find the used_key there for
( const k in data . result . keys )
{ if
( data . result . keys [ k ] . public_key
=== publicKey )
{ // Ensure the key is full access, meaning the user had to sign // the transaction through the wallet return data . result . keys
[ k ] . access_key . permission
==
"FullAccess" } }
return
false

// didn't find it }

// Aux method async
function
fetch_all_user_keys ( { accountId } )
{ const keys =
await
fetch ( "https://rpc.testnet.near.org" , { method :
'post' , headers :
{
'Content-Type' :
'application/json; charset=utf-8'
} , body :
{"jsonrpc":"2.0", "method":"query", "params":["access_key/ { accountId } ", "", "id":1} } } ) . then ( data
=> data . json ( ) ) . then ( result
=> result ) return keys }

module . exports
=
{ authenticate , verifyFullKeyBelongsToUser , verifySignature } ;Edit this page Last updated on Feb 9, 2024 by gagdiez Was

```

this page helpful? Yes No

[Previous Integrating Components](#) [Next What are Primitives?](#)