

Sign data

You can use the following RPC methods to request cryptographic signatures from users:

- [eth_signTypedData_v4](#)
 - Use this method to request the most human-readable signatures that are efficient to process on-chain.
- We recommend this for most use cases.
- [personal_sign](#)
 - Use this method for the easiest way to request human-readable signatures that don't need to be efficiently processed on-chain.

caution [eth_sign](#) is deprecated. note MetaMask supports signing transactions using Trezor and Ledger hardware wallets. These wallets only support signing data using [personal_sign](#). If you can't log in to a dapp when using a Ledger or Trezor, the dapp might be requesting you to sign data using an unsupported method, in which case we recommend using your standard MetaMask account.

Use [eth_signTypedData_v4](#)

[eth_signTypedData_v4](#) provides the most human-readable signatures that are efficient to process on-chain. It follows the [EIP-712](#) specification to allow users to sign typed structured data that can be verified on-chain. It renders the structured data as usefully as possible to the user (for example, displaying known account names in place of addresses).

An [eth_signTypedData_v4](#) payload uses a standard format of encoding structs, but has a different format for the top-level struct that is signed, which includes some metadata about the verifying contract to provide replay protection of these signatures between different contract instances.

We recommend using [eth-sig-util](#) to generate and validate signatures. You can use [eip712-codegen](#) to generate most of the Solidity required to verify these signatures on-chain. It currently doesn't generate the top-level struct verification code, so you must write that part manually. See [this example implementation](#).

caution Since the top-level struct type's name and the `domain.name` are presented to the user prominently in the confirmation, consider your contract name, the top-level struct name, and the struct keys to be a user-facing security interface. Ensure your contract is as readable as possible to the user.

Example

The following is an example of using [eth_signTypedData_v4](#) with MetaMask:

```
index.js signTypedDataV4Button . addEventListener ( "click" ,
  async
  function
  ( event )
  { event . preventDefault ( ) ;
    // eth_signTypedData_v4 parameters. All of these parameters affect the resulting signature. const msgParams =
    JSON . stringify ( { domain :
      { // This defines the network, in this case, Mainnet. chainId :
        1 , // Give a user-friendly name to the specific contract you're signing for. name :
        "Ether Mail" , // Add a verifying contract to make sure you're establishing contracts with the proper entity. verifyingContract :
        "0xCcCCccccCCCCcCCCCCCcCcCccCcCCCcCcccccccC" , // This identifies the latest version. version :
        "1" , } ,
    // This defines the message you're proposing the user to sign, is dapp-specific, and contains // anything you want. There are
    no required fields. Be as explicit as possible when building out // the message schema. message :
```

type :

```
"string"
} , {
name :
"members" ,
type :
"Person[]"
} , ] , // Refer to primaryType. Mail :
[ {
name :
"from" ,
type :
"Person"
} , {
name :
"to" ,
type :
"Person[]"
} , {
name :
"contents" ,
type :
"string"
} , ] , // Not an EIP712Domain definition. Person :
[ {
name :
"name" ,
type :
"string"
} , {
name :
"wallets" ,
type :
"address[]"
} , ] , } , } ) ;
var
from
=
```

```

await web3 . eth . getAccounts ( ) ;

var params =

[ from [ 0 ] , msgParams ] ; var method =

"eth_signTypedData_v4" ;

provider // Or window.ethereum if you don't support EIP-6963. . sendAsync ( { method , params , from :

from [ 0 ] , } , function

( err , result )

{ if

( err )

return

console . dir ( err ) ; if

( result . error )

{ alert ( result . error . message ) ; } if

( result . error )

return

console . error ( "ERROR" , result ) ; console . log ( "TYPED SIGNED:"

+

JSON . stringify ( result . result ) ) ;

const recovered = sigUtil . recoverTypedSignature_v4 ( { data :

JSON . parse ( msgParams ) , sig : result . result , } ) ;

if

( ethUtil . toChecksumAddress ( recovered )

=== ethUtil . toChecksumAddress ( from ) )

{ alert ( "Successfully recovered signer as "

+

from ) ; }

else

{ alert ( "Failed to verify signer when comparing "

+ result +

" to "

+

from ) ; } } ) ; } ) ; The following HTML displays a sign button:

index.html < h3

    Sign typed data v4 </ h3

    < button

```

type

" button "

id

```
" signTypedDataV4Button "
```

```
eth_signTypedData_v4 </ button
```

See the[live example](#) and[test dapp source code](#) for more information.

Usepersonal_sign

[personal_sign](#) is the easiest way to request human-readable signatures that don't need to be efficiently processed on-chain. It's often used for signature challenges that are authenticated on a web server, such as[Sign-In with Ethereum](#).

Some other signers implement `personal_sign` as `eth_sign`, because the Go Ethereum client changed the behavior of their `eth_sign` method. Because MetaMask supports existing applications, MetaMask implements both `personal_sign` and `eth_sign`. You might need to check what method your supported signers use for a given implementation.

important * Don't use this method to display binary data, because the user wouldn't be able to understand what * they're agreeing to. * If using this method for a signature challenge, think about what would prevent a phisher from * reusing the same challenge and impersonating your site. * Add text referring to your domain, or the current time, so the user can easily verify if this * challenge is legitimate.

Example

The following is an example of using `personal_sign` with MetaMask:

```
index.js personalSignButton . addEventListener ( "click" ,
```

```
async
```

```
function
```

```
( event )
```

```
{ event . preventDefault ( ) ; const exampleMessage =
```

```
"Example personal_sign message." ; try
```

```
{ const
```

```
from
```

```
= accounts [ 0 ] ; // For historical reasons, you must submit the message to sign in hex-encoded UTF-8. // This uses a Node.js-style buffer shim in the browser. const msg =
```

```
` 0x { Buffer . from ( exampleMessage ,
```

```
"utf8" ) . toString ( "hex" ) } ` ; const sign =
```

```
await ethereum . request ( { method :
```

```
"personal_sign" , params :
```

```
[ msg ,
```

```
from ] , } ) ; personalSignResult . innerHTML
```

```
= sign ; personalSignVerify . disabled
```

```
=
```

```
false ; }
```

```
catch
```

```
( err )
```

```
{ console . error ( err ) ; personalSign . innerHTML
```

=

Error: { err . message } ; } }) ; The following HTML displays a sign button:

index.html < h3

Personal sign </ h3

< button

type

" button "

id

" personalSignButton "

personal_sign </ button

personal_sign prepends the message with\x19Ethereum Signed Message:\n before hashing and signing it.

See the[live example](#) and[test dapp source code](#) for more information.

[Edit this page](#)