How To Implement Gas-Less Transactions on Ethereum

Unlocking Ethereum for the masses

Alberto Cuesta Cañada

Follow

Coinmonks

--

2

Listen

Share

Everyone talks about “gas-less” Ethereum transactions because no one likes paying for gas. But the Ethereum network runs precisely because transactions are paid for. Then, how can you have “gas-less” anything? What is this sorcery?

In this article, I’m going to show how to use the patterns behind “gas-less” transactions. You will discover that although there is no such thing as a free lunch in Ethereum, you can shift gas costs in interesting ways.

By applying the knowledge from this article, your users will save on gas, will enjoy a better UX, and even build novel delegation patterns into your smart contracts.

But wait! There is more! For your convenience, I’ve put all the tools needed in this repository. So now the barrier for you to implement “gas-less” tokens is suddenly much lower.

Let’s get nerdy.

Background

I have to confess that even if I know how to implement “gas-less” transactions into smart contracts, I know very little about the cryptography that makes them possible. That wasn’t a major obstacle for me, so it shouldn’t be for you either.

As far as I know, my private key is used to sign the transactions I send to Ethereum, and some cryptography magic is used to identify me as msg.sender

. That underpins all access control in Ethereum.

The sorcery behind “gas-less” transactions is that I can produce a signature with my private key and the smart contract transaction that I want executed.

The signature would be produced off-chain, without spending anything on gas. Then I could give this signature to someone else to execute the transaction on my behalf, with their gas.

The function that the signature is for will usually be a regular function, but extended with additional signature parameters. For example in dai.sol we have the approve

function:

We also have the permit

function, which does the same as approve

but takes a signature as a parameter.

Don’t worry about all those extra parameters, we’ll get to them. What you need to pay attention to is what both functions do with the allowance

mapping:

If you use approve

, you allow spender

to use up to wad

of your tokens.

If you give a valid signature to someone, that someone can call permit

to allow spender

using your tokens.

So basically, the pattern behind “gas-less” transactions is to craft a signature that you can give to someone, so that they can safely execute a special transaction. It’s like giving permission to someone to execute a function.

It is a delegation pattern.

The standards

If you are like me, the first thing you will do is to dive into the code. I immediately noticed this comment:

With that, I went down the rabbit hole, and got hopelessly lost. Now that I understand it, I can explain it in plain terms.

EIP712 describes how to build signatures for functions, in a generic way. Other EIPs describe how to apply EIP712 to specific use cases. For example EIP2612 describes how to use EIP712 signatures for a function called permit

which should have the same functionality as approve

in an ERC20 token.

If you just want to implement a signature function that has been done before, like adding signature approves to your own MetaCoin, then you can read EIP2612 and you will be well on your way. You can even inherit from a contract implementing it and limit the stress in your life.

In this article we will investigate an implementation of “gas-less” transactions in dai.sol, which will make things clear. The dai.sol implementation happened before EIP2612 and is slightly different. That will not be a problem.

Signature composition

An early implementation of EIP712 signatures can be found in dai.sol. It allows dai holders to approve transfer transactions by calculating an off-chain signature and giving it to the spender

, instead of calling approve

themselves.

It includes four elements:

- A DOMAIN_SEPARATOR

.

- A PERMIT_TYPEHASH

.

- A nonces

variable.

- A permit

function.

This is the DOMAIN_SEPARATOR

, with related variables:

The DOMAIN_SEPARATOR

is nothing more than a hash that uniquely identifies a smart contract. It is built from a string denoting it as an EIP712 Domain, the name of the token contract, the version, the chainId in case it changes, and the address that the contract is deployed at.

All that information is hashed on the constructor into the DOMAIN_SEPARATOR

variable, which will have to be used by the holder when creating the signature, and will need to match when executing permit

. That ensures that a signature is valid for one contract only.

This is the PERMIT_TYPEHASH

:

The PERMIT_TYPEHASH

is the hash of the function name (capitalized) and all the parameters including type and name. It's purpose is to clearly identify which function is the signature for.

The signature will be processed in the permit

function, and if the PERMIT_TYPEHASH

used was not for this specific function, it will revert. This makes sure that a signature is only used for the intended function.

Then there is the nonces

mapping:

This mapping registers how many signatures have been used for a particular holder. When creating the signature, a nonces

value needs to be included. When executing permit

, the nonce included must exactly match the number of signatures that have been used so far for that holder. This ensures that each signature is used only once.

All these three conditions together, the PERMIT_TYPEHASH

, the DOMAIN_SEPARATOR

, and the nonce

, make sure that each signature is used only for the intended contract, the intended function, and only once.

Now let's see how the signature would be processed in the smart contract.

The permit function

permit

is the dai.sol function that allows using signatures to modify the allowance

of holder

towards spender

.

As you can see, there are a lot of parameters there. They are all the parameters needed to compute the signature, plus v

, r

and s

which are the signature itself.

It seems silly that you need the parameters that were used to create the signature, but you do. The only thing that you can recover from the signature is the address that created it, nothing more. We will use all the parameters and the recovered address to ensure the signature is valid.

First we calculate a digest

using all the parameters that we will need to ensure safety. The holder

will need to calculate the exact same digest off-chain, as part of the signature creation:

Using ecrecover

and the v,r,s

signature we can recover an address. If it is the address of the holder, we know that all the parameters match (DOMAIN_SEPARATOR

, PERMIT_TYPEHASH

, nonce

, holder

, spender

, expiry

, and allowed

. If anything is off, the signature is rejected:

A word of caution here. There are many parameters that go into a signature, some of them obscure like the chainId

(part of the DOMAIN_SEPARATOR

). Any of them being off will cause the signature being rejected, which guarantees that debugging off-chain signatures will be difficult. You have been warned.

Now we know that the holder

approved this function call. Next we will certify that the signature is not being abused. First, we check that the current time is before the expiry

, this allows permits to be held only for a specific period.

We also check that a signature with that nonce

hasn't been used yet, so that each signature can be used only once.

And we are through! dai.sol maxes out the allowance

of holder

towards spender

, emits an event, and that's it.

The dai.sol contract has a binary approach towards allowance

, in the repository provided you'll find a more traditional behavior.

Creating the signature off-chain

Creating the signature is not for the faint of heart, but with a bit of practice and persistence it can be mastered. We will replicate what the smart contract does in permit

in three steps:

1. Generate the DOMAIN_SEPARATOR
2. Generate the digest
3. Create the transaction signature

The following function will create the DOMAIN_SEPARATOR

. It is the same code as in the dai.sol constructor, but in JavaScript and using keccak256

, defaultAbiCoder

and toUtfBytes

from ethers.js. It needs the token name and deployment address, along with the chainId

. It assumes the token version to be "1".

The following function will create a digest for a specific permit

call. Note that the holder

, spender

, nonce

and expiry

are passed on as arguments. It also passes an approve.allowed

argument for clarity, although you could just set it always to true

, otherwise the signature will be rejected and what would be the point? The PERMIT_TYPEHASH

we just copied it from dai.sol.

Once we have a digest, signing it is relatively easy. We just use ecsign

from ethereumjs-util after removing the 0x

prefix from the digest. Note that we need the user private key to do this.

In the code, we would call these functions as follows:

Note how the call to permit

reuses all the parameters that were used to create the digest, before it was signed. Only in that case the signature would be valid.

Note as well that the only two transactions in this snippet are being called by user2

. user1

is the holder

, and is the one that created the digest and signed it. However, user1

didn't spend any gas doing so.

user1

gave the signature to user2

, which used it to execute both the permit

and the transferFrom

that user1

allowed.

From the point of view of user1

, it was a "gas-less" transaction. He didn't spend a wei.

Conclusion

This article shows how to use "gas-less" transactions, clarifying that "gas-less" actually means passing the gas cost to someone else. To do that we need a function in a smart contract that is ready to deal with pre-signed transactions, and a good deal of data manipulation to make everything safe.

However, there are significant gains from using this pattern, and for that reason it is widely used. Signatures allow passing the transaction gas cost from the user to the service provider, eliminating a considerable barrier in many cases. It also allows for the implementation of more advanced delegation patterns, often with considerable UX improvements.

A repository has been provided for you to get started. Please use it, and pleasecontinue the conversation.

Very special thanks to Georgios Konstantopoulos, who taught me all I know about this pattern.

Also, Read

- Crypto Telegram Signals | Crypto Trading Bot
- Best Crypto Exchange | Best Crypto Exchange in India
- Best Crypto APIs for Developers
- Buy Bitcoin India | Pionex Review | Crypto Trading bots
- NGRAVE ZERO review | Phemex Review | PrimeXBT Review
- Best Blockchain Analysis Tools | Earn Bitcoin
- Cloudbet Casino Review | Ignition Casino Review
- Crypto arbitrage Guide | How to Short Bitcoin
- Best Crypto Lending Platform
- Free Crypto Signals | Crypto Trading Bots
- An ultimate guide to Leveraged Token