@elnaril
Elnaril

Python: How to Use Permit2 with the Uniswap Universal Router

This is the long-awaited and ardently demanded tutorial about the **Permit2** contract and how to use it with the Uniswap Universal Router (UR) to swap or sell tokens. It is also the continuation of the introductory tutorial: **How to Buy a Token on the Uniswap Universal Router with Python**.

A Bit of Context

After ERC-20 (2015), the standard way to allow a protocol to use your tokens was to `approve` it to the token contract and give it an `allowance`, through a transaction.

Then, ERC-2612 (2020) introduced the `permit` concept as a standard to give this allowance using a signed message (thanks to the EIP-712 data signing standard) instead of the approval transaction. This EIP brought some nice benefits like:

- Simplification: less approvals
- Gas cost: reduced cost, or even no gas at all
- Security: approvals include an expiration timestamp and a nonce

But because it is built on top of ERC-20, only new (or upgradeable) tokens could enjoy them …

Then, Uniswap released the Permit2 contract along with the Universal Router (November 2022), which allows signature-based approvals for all tokens, **even if they don't implement ERC-2612.**

Tutorial Scope and Target

The Permit2 and Universal Router contracts have many features, but we'll focus only on how to leverage both contracts to swap tokens using signed approvals. More specifically, we're going to learn how to use the UR ==**PERMIT2_PERMIT**== function.

We finished the introductory tutorial after buying some UNI tokens, so we'll start from there, and we'll swap them for some USDT.
:::info By the end of this tutorial, you will be able to create a transaction to **sell** some tokens from the latest Uniswap router with Python, using **Permit2 signature-based approvals**.
:::

Prerequisites

This tutorial is intended for developers who are comfortable with Python, the basics of Ethereum (contracts, ERC20, transactions, gas, etc.), and building transactions with web3.py.
If not already done, it is also strongly recommended to read the introductory tutorial first (**How to Buy a Token on the Uniswap Universal Router with Python**) to be familiar with the tools, libraries, and setup we're going to use here as well.

Libraries, Tools and Setup

In this tutorial, we're going to use the same libraries and tools as we did in the introductory one, so you can refer to it for installation and setup:

- Web3.py
- Uniswap Universal Router Codec
- Python virtual environment
- Ganache CLI
- Node.js and npm

:::tip You don't want to learn to swap tokens with actual ones. So you can use either a testnet like **Sepolia** or a local fork created with Ganache, which is the solution I'll use in this tutorial.
:::

New versions of the Web3.py library and UR SDK have been released in the meantime, so be sure you have the latest ones with the following command:

```
$ pip freeze
```

You should get something like:

```
uniswap-universal-router-decoder==0.9.1
web3==6.11.1
```

if you have older versions, just update the libraries as usual:

```
$ pip install -U uniswap-universal-router-decoder web3
```

Launching Ganache

We're going to launch Ganache to fork Ethereum locally at block number 17365005, as we did previously. We're going to use the same private key and give it 100 ETH (= `100 * 10**18` wei, which is `0x56bc75e2d63100000` in hexadecimal).
My endpoint address is stored in the environment variable `$RPC_ENDPOINT`, so the command is:

```
$ ganache --fork.url=$RPC_ENDPOINT --wallet.accounts=0x6c9803151aa5cf420e98dd1afb9db96f0510918cb2249e9858519d47777125c9,0x56bc75e2
```

:::warning Do **NOT** use the private keys/accounts from these tutorials (or from anywhere) on Mainnet! Everyone could access your fund!
Also, if you see some tokens in these accounts on Mainnet (or in any account whose private key is on the web), **NEVER** try to get them: a sweeper bot will steal your gas!
:::

Getting some UNI Tokens

We're going to re-use the script buy_token.py from the previous tutorial and append our new code to have a fully independent script to launch. Make sure the script works and gives you `368.41333521045374` UNI.

Swapping tokens

And now, the moment you've been waiting for: selling our UNI tokens using the Uniswap Universal Router and the Permit2 contract!

Step 1: Approving the Permit2 contract to the UNI one

This step is similar to the ERC-20 standard, *except that you approve the Permit2 contract instead of the router*. It is the step that allows all tokens, even those that don't implement ERC-2612, to use signature-based approvals.

```
# approve Permit2 to UNI
permit2_address = Web3.to_checksum_address("0x000000000022D473030F116dDEE9F6B43aC78BA3")
permit2_allowance = 2**256 - 1  # max
```

```
contract_function = uni_contract.functions.approve(
permit2_address,
permit2_allowance
)
trx_params = contract_function.build_transaction(
{
"from": account.address,
"gas": 500_000,
"maxPriorityFeePerGas": w3.eth.max_priority_fee,
"maxFeePerGas": 100 * 10**9,
"type": '0x2',
"chainId": chain_id,
"value": 0,
"nonce": w3.eth.get_transaction_count(account.address),
}
)
raw_transaction = w3.eth.account.sign_transaction(trx_params, account.key).rawTransaction
trx_hash = w3.eth.send_raw_transaction(raw_transaction)
print(f"Permit2 UNI approve trx hash: {trx_hash.hex()}")
```

In this tutorial, we give the maximum allowance (2**256 - 1 wei), but depending on your use case, you might want to give less on Mainnet.

Checking the allowance we have just approved:

```
print(
"Permit2 UNI allowance:",
uni_contract.functions.allowance(account.address, permit2_address).call(),
)
```

You should get:

```
Permit2 UNI allowance: 79228162514264337593543950335
```

Step 2: Building and Signing the permit message

Now, we're ready to build the swap/sell transaction with a signature-based approval. Let's start with the permit message.

a. Permit2 nonce and allowance

For security purposes, the message you need to sign contains a nonce. It is incremented for each permit message you sign. The nonce depends on your account address and on the token and universal router addresses.

To know the current Permit2 nonce, allowance, and expiration:

```
permit2_abi = '[{"inputs":[{ ... "type":"function"}]'  # truncated
permit2_contract = w3.eth.contract(address=permit2_address, abi=permit2_abi)
p2_amount, p2_expiration, p2_nonce = permit2_contract.functions.allowance(
account.address,
uni_address,
ur_address
).call()
print(
"p2_amount, p2_expiration, p2_nonce: ",
p2_amount,
p2_expiration,
p2_nonce,
)
```

For readability sake, I've truncated most of the abi, but you can get it from Etherscan, as explained in the previous tutorial. If you have a doubt, you can check the full tutorial code.

We haven't yet given any allowance for Permit2 to the UNI contract, so we expect only zeros:

```
permit2_amount, permit2_expiration, permit2_nonce:  0 0 0
```

b. Building the signable message

The approval message is built as follows:

```
# permit message
allowance_amount = 2**160 - 1  # max/infinite
permit_data, signable_message = codec.create_permit2_signable_message(
uni_address,
allowance_amount,
codec.get_default_expiration(),  # 30 days
p2_nonce,
ur_address,
codec.get_default_deadline(),  # 180 seconds
chain_id,
)
```

Let's breakdown the command:

- `allowance_amount` is the amount you're happy to approve. Here, we set it to the infinite value: `2**160 - 1`
- `codec.get_default_expiration()` gives the timestamp after which the allowance is not valid anymore. The codec default is at 30 days, but you can set whatever you want in the future: you could set 24h like that: `codec.get_default_expiration(24 * 3600)`
- `permit2_nonce`: the Permit2 nonce we just got at the previous step.
- `codec.get_default_deadline()` : the timestamp after which the transaction will not be valid anymore. Default is 180s.
- `chain_id`: 1 for Ethereum, 1337 for Ganache, …
- `permit_data` contains the permit message details (basically, the `codec.create_permit2_signable_message()` arguments)
- `signable_message` is the EIP-172 encoded version of the permit message.

c. Signing the message

Signing the message is as simple as:

```
signed_message = account.sign_message(signable_message)
```

Step 3: The swap transaction

a. Encoding the input data

The swap part in the transaction we're going to send to the Universal Router is very similar to what we learned in the previous tutorial. Now could be a good moment to look into it again and refresh our memory about the `v2_swap_exact_in()` method because we're using it again here:

```
# Building the Swap to sell UNI for USDT
usdt_address = Web3.to_checksum_address("0xdac17f958d2ee523a2206206994597c13d831ec7")
usdt_abi = '[{"constant" ... :"Unpause","type":"event"}]'  # truncated
usdt_contract = w3.eth.contract(address=usdt_address, abi=usdt_abi)
amount_in = 100 * 10**18
min_amount_out = 415 * 10**6
path = [uni_address, usdt_address]
encoded_input = (
codec
.encode
.chain()
.permit2_permit(permit_data, signed_message)
.v2_swap_exact_in(
FunctionRecipient.SENDER,
amount_in,
min_amount_out,
path,
payer_is_sender=True,
)
.build(codec.get_default_deadline())
)
```

Again, the ABI is truncated: you can get it from Etherscan or from the tutorial full code.

Let's breakdown the command:

- `codec`, `encode`, `chain()`, `v2_swap_exact_in()` and `build()` have already been explained in the introductory tutorial.
- `permit2_permit()`: Encode the call to the Universal router function **PERMIT2_PERMIT**, with the previously built arguments: `permit_data` and `signed_message`.
- `min_amount_out`: at block `17365005`, the price of UNI was around $4.19, so we gave a bit of room and asked for a minimum of 415 USDT for our 100 UNI.

b. Building and sending the transaction

And the transaction is built and sent as usual:

```
trx_params = {
"from": account.address,
"to": ur_address,
"gas": 500_000,
"maxPriorityFeePerGas": w3.eth.max_priority_fee,
"maxFeePerGas": 100 * 10**9,
"type": '0x2',
"chainId": chain_id,
"value": 0,
"nonce": w3.eth.get_transaction_count(account.address),
"data": encoded_input,
}
raw_transaction = w3.eth.account.sign_transaction(trx_params, account.key).rawTransaction
trx_hash = w3.eth.send_raw_transaction(raw_transaction)
print(f"Trx Hash: {trx_hash.hex()}")
```

c. Checking the balances

If all goes well, you should have 100 UNI less and a positive USDT balance:

```
# Checking the balances
uni_balance = uni_contract.functions.balanceOf(account.address).call()
print("UNI Balance:", uni_balance / 10**18, "UNI")
usdt_balance = usdt_contract.functions.balanceOf(account.address).call()
print("USDT Balance:", usdt_balance / 10**6, "USDT")
```

```
UNI Balance: 268.41333521045374 UNI
USDT Balance: 419.503665 USDT
```

d. Checking the new Permit2 allowance

```
# Checking the new Permit2 allowance
p2_amount, p2_expiration, p2_nonce = permit2_contract.functions.allowance(
account.address,
uni_address,
ur_address
).call()
print(
"p2_amount, p2_expiration, p2_nonce: ",
p2_amount,
p2_expiration,
p2_nonce,
)
```

Now, you should get some values similar to:

```
p2_amount, p2_expiration, p2_nonce:  1461501637330902918203684832716283019655932542975 1700722499 1
```

:::tip As long as `p2_amount > 0` and `p2_expiration` is in the future, you can swap the USDT token; you don’t have to add the `permit2_permit(permit_data, signed_message)` method when building your next transaction!
:::

Conclusion

Using the PERMIT2_PERMIT function is the way to authorize the Uniswap Universal Router to swap any ERC-20 tokens, even if they don’t support ERC-2612. The great benefit, compared to the ERC-20 standard, is that you don’t have to approve each and every protocol to the USDT contract if they use the Permit2 contract as well. You just need to include a signed permit message in your transaction.
:::info If you made it up to here, congrats !!!
You have learnt:

- How to use the Permit2 contract with the Universal Router to swap a token
- How to encode and sign a Permit2 permit message

In addition, you have:

- Acquired a better understanding of several important protocols and EIPs

- Practised what you learnt in the previous tutorial

:::

Closing words and disclaimer

Now that you have a more advanced knowledge of the UR codec, you can continue to play with the other functions it supports.

Disclaimer: I'm the author and maintainer of the Python Universal Router SDK. This open-source library is a work in progress and does not support all UR functions yet. If you ever experience any issues with it, please open a ticket here. And feel free to share what other UR functions you would like to be supported by this SDK.

Tags: programming,python,python-programming,python-tutorials,uniswap,defi,cryptocurrency,blockchain-development

Published On: Tue Oct 24 2023 13:01:03 GMT+0000 (Coordinated Universal Time)!

View Story Live at: https://hackernoon.com/python-how-to-use-permit2-with-the-uniswap-universal-router!