How to implement Uniswap Permit2 in your protocol

Usually when interacting with a protocol that is transferring users `ERC20` tokens, the user must first approve it by calling `approve` on the `ERC20` token contract. This means that for every new application that user interacts with, he must make 2 transactions (one to `approve` and the second one to call the smart contract he wants to interact with).

This is why a few years ago `EIP-2612: Permit Extension for EIP-20 Signed Approvals` was introduced. `EIP-2612` tokens implement a `permit()` function, which allows users to set the `allowance` using a signed message. This way user can give this signed message to a contract he wants to interact with, and the contract can use it to call `permit()` and set the `allowance`. However, there are still a lot of tokens that don't implement this standard. **Permit2** is the Uniswap Labs solution to this problem.

What does it do?

`Permit2` system manages all users allowances in one contract. Instead of setting the allowance for each protocol separately, user sets the allowance only for the `Permit2` contract. The protocol contract that user wants to interact with, can then call the `Permit2` contract to transfer the tokens. The tokens can be transferred if allowance is set or a `permit` message signed by the user is provided. The allowance can be set by the user directly or through a Signature Based Approval. There are also features like Batch Transfers/Approvals, Expiring Approvals, Safe Arbitrary Data Verification, Signature Verification for Contracts, Batch Revoke Allowances, Replay Protection. You can read more about these features on their github.

How to implement it in your protocol?

First we will go over how you can implement the `Permit2` system in your smart contracts and then how to create and sign a `permit` message.

There are two main ways you can use the `Permit2` system to transfer tokens:

1. Transferring using allowance – User signs a permit message that the contract then uses to set the allowance (or he sets the allowance

himself) and transfer the tokens. This requires 2 calls to `Permit2`. Both can be executed from the contract in one transaction.

2. Transferring using signature – User sings a permit message that the contact then uses to transfer the tokens without the need for allowance. This requires only 1 call to `Permit2`.

1. Transferring using signature

As stated, with Signature Based Token Transfer, there is no need for allowance. User signs a message that the contract then uses to transfer tokens. The signature is valid only for the duration of the transaction in which it is spent.

To execute a transfer using signature, we have to use one of the functions from ISignatureTransfer. There are 4 functions that can be used. 2 of which are named `permitTransferFrom` and 2 named `permitWitnessTransferFrom`. The reason there are 2 of each is that one is used for a single token transfer and one for bulk, which you can see by looking at `SignatureTransferDetails` parameter. The difference between `permitTransferFrom` and `permitWitnessTransferFrom` is that the second one includes extra data provided by the caller to verify signature over. We will start with `permitTransferFrom`, since it is the simplest one.

```
function permitTransferFrom(
      PermitTransferFrom memory permit,
      SignatureTransferDetails calldata transferDetails,
      address owner,
      bytes calldata signature
) external;
```

As you can see the function requires 4 parameters:

- `permit` – the `PermitTransferFrom` message that includes the following data:
  - what token we are permitting to be transferred,
  - how many tokens are we permitting to be transferred,
  - deadline when the permit expires,
  - and nonce to prevent signature replays
- `transferDetails` – details about the token transfer including:
  - address of token receiver
  - amount of tokens to be transferred
- `owner` – token owner
- `signature` – signed `permit` message hash by the `owner`

Now let's implement it in our contract. To interact with `Permit2` we will need [ISignatureTransfer] interface that has everything needed for signature based token transfers. Copy the interface to your codebase and import it. (I have created `interfaces` directory and copied the interface there.)

```
import {ISignatureTransfer} from "./interfaces/ISignatureTransfer.sol";
```

We will need the `Permit2` contract address, so add it to the contract.

```
ISignatureTransfer public immutable PERMIT2;

constructor(ISignatureTransfer _permit) {
  PERMIT2 = _permit;
}
```

Next, create a function that needs users permission to transfer tokens. For this example, we will be creating a `deposit` function that will transfer users tokens to the contract and update his balance.

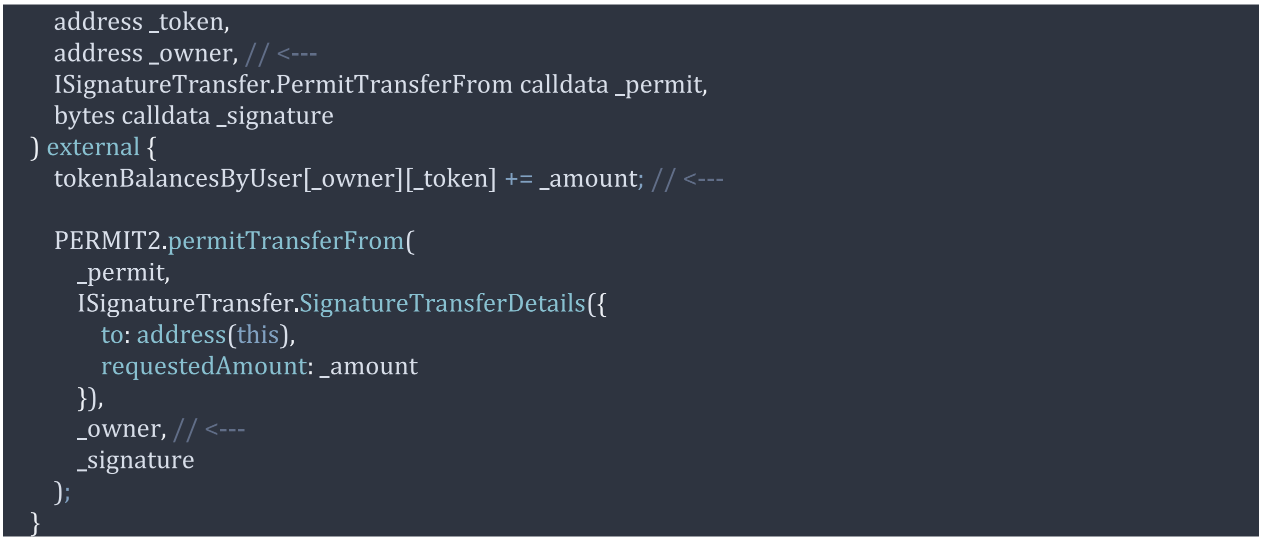
```
  mapping (address => mapping (address => uint256)) public tokenBalancesByUser;

  function deposit(
    uint256 _amount,
    address _token,
    ISignatureTransfer.PermitTransferFrom calldata _permit,
    bytes calldata _signature
  ) external {
    // update users balance
    tokenBalancesByUser[msg.sender][_token] += _amount;

    PERMIT2.permitTransferFrom(
      // The permit message.
      _permit,
      // The transfer recipient and amount.
      ISignatureTransfer.SignatureTransferDetails({
        to: address(this),
        requestedAmount: _amount
      }),
      // Owner of the tokens and signer of the message.
      msg.sender,
      // The packed signature that was the result of signing
      // the EIP712 hash of `_permit`.
      _signature
    );
  }
```

As you can see, the `owner` (3rd param in `permitTransferFrom`) is `msg.sender`. This means only the owner can use his own signature. If your protocol needs the caller to be the owner himself, this is the way to do it. But you might want to enable a third party to execute deposit in the name of the owner. In that case you could do it like this:

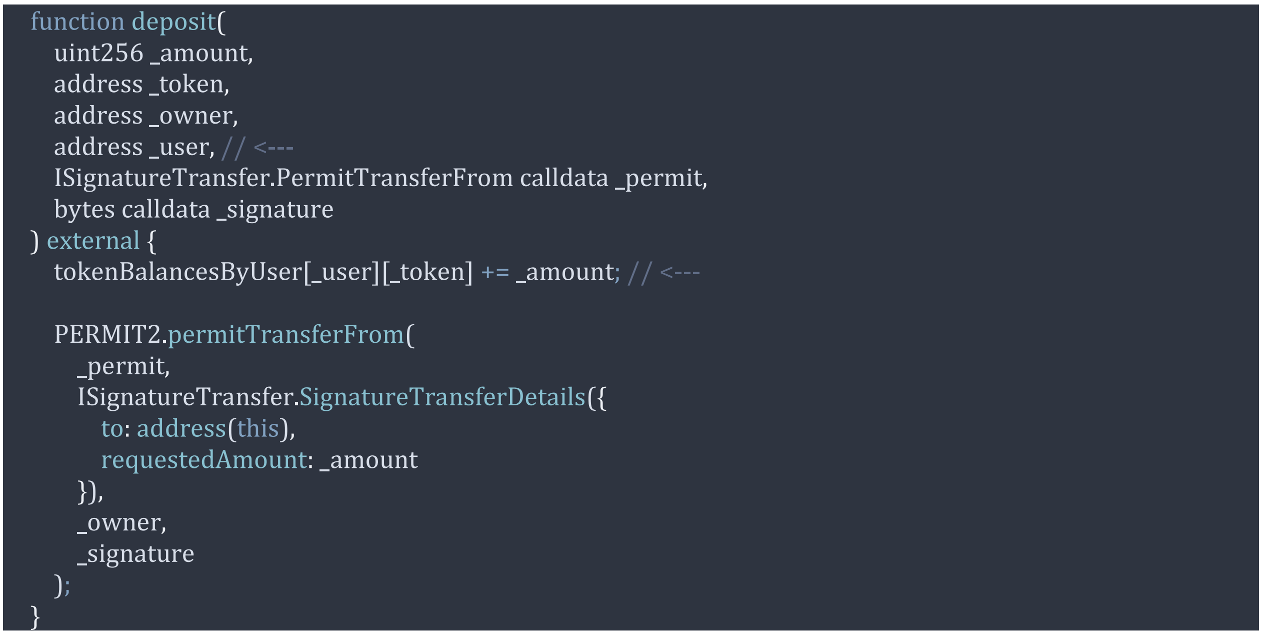
```
  function deposit(
    uint256 _amount,
```



```
    address _token,
    address _owner, // <---
    ISignatureTransfer.PermitTransferFrom calldata _permit,
    bytes calldata _signature
) external {
    tokenBalancesByUser[_owner][_token] += _amount; // <---

    PERMIT2.permitTransferFrom(
        _permit,
        ISignatureTransfer.SignatureTransferDetails({
            to: address(this),
            requestedAmount: _amount
        }),
        _owner, // <---
        _signature
    );
}
```

It is important to make sure your protocol is safe by allowing usage of others signatures. In the above code it is safe, because the owners signature can be used only to deposit the tokens to the owner himself. Now lets create an example where that would not be the case. In this example, we will allow users to deposit tokens and give them to others.



```
function deposit(
    uint256 _amount,
    address _token,
    address _owner,
    address _user, // <---
    ISignatureTransfer.PermitTransferFrom calldata _permit,
    bytes calldata _signature
) external {
    tokenBalancesByUser[_user][_token] += _amount; // <---

    PERMIT2.permitTransferFrom(
        _permit,
        ISignatureTransfer.SignatureTransferDetails({
            to: address(this),
            requestedAmount: _amount
        }),
        _owner,
        _signature
    );
}
```

What can go wrong here? The `owner` can sign a message to transfer X tokens, and give them to user A. But the caller can use `owner` signature and give the tokens to the balance of `_user`, which can be any arbitrary address not just user A. To fix this, we can either disable using others signatures by replacing `_owner` variable with `msg.sender`, or we could use `permitWitnessTransferFrom`.

With `permitWitnessTransferFrom` we can include the user address that we want to give the tokens to in the signature. That way we can make sure the signature is used only to give the tokens to user A.

`permitWitnessTransferFrom` accepts 2 extra parameters:

- `witness` – The data that we want to include in user signature
- `witnessTypeString` – The EIP-712 type definition for remaining string stub of the typehash

For our `witness` we need to create a struct that will hold the data we want to validate.

```
struct Witness {
  // Address of the user that signer is giving the tokens to
  address user;
}
```

Now we need to create a `witnessTypeString` that will allow `Permit2` to produce the right hash. If you are not familiar with EIP-712, I highly recommend checking it out, since it describes the rules for typed structured data hashing and signing.

Let's see the first part of the type string that our `winessTypeString` will be appended to.

```
"PermitWitnessTransferFrom(TokenPermissions permitted,address spender,uint256 nonce,uint256 deadline,"
```

You can see that it ends with a comma, because it is expecting us to add the information about our witness type to it. So let's add "Witness witness" and closing brackets ")".

```
"PermitWitnessTransferFrom(TokenPermissions permitted,address spender,uint256 nonce,uint256 deadline,Witness witness)"
```

Cool, but there is still something missing. We can see that we have two struct types in this string `TokenPermissions` and our `Witness` descriptions. As described in EIP-712, referenced struct types must be appended to the end of the type string.

```
"PermitWitnessTransferFrom(TokenPermissions permitted,address spender,uint256 nonce,uint256 deadline,Witness witness)TokenPermissions(address token,uint256 amount)Witness(address user)"
```

Notice, that `Witness` type comes after `TokenPermissions`. This is because they must be sorted by name. If we would name the witness struct `MyWitness`, we would have to put it before `TokenPermissions`.

Okay, now we know how the full type string has to look like, but we only need the part that we added. So add it to the contract.

```
string private constant WITNESS_TYPE_STRING = "Witness witness)TokenPermissions(address token,uint256 amount)Witness(address user)";
```

We will also need to hash the witness data, for which we need to create a type hash.

```
bytes32 private constant WITNESS_TYPEHASH = keccak256("Witness(address user)");
```

Next, recreate the `deposit` function to use the witness data.

```
  function deposit(
    uint256 _amount,
    address _token,
    address _owner,
    address _user,
    ISignatureTransfer.PermitTransferFrom calldata _permit,
    bytes calldata _signature
  ) external {
    tokenBalancesByUser[_user][_token] += _amount;

    PERMIT2.permitWitnessTransferFrom(
      _permit,
      ISignatureTransfer.SignatureTransferDetails({
        to: address(this),
        requestedAmount: _amount
      }),
      _owner,
      // witness
      keccak256(abi.encode(WITNESS_TYPEHASH,Witness(_user))),
      // witnessTypeString,
      WITNESS_TYPE_STRING,
      _signature
    );
  }
```

Great, now the permit message signer has the control over who his tokens will be given to.

2. Transferring using allowance

Transferring using allowance is similar to `EIP-2612`, but with the addition that the user can add an expiration to the approval.

In short, it goes like this: User signs a message that the contract can use to set the allowance by calling `Permit2.permit`. When the allowance is set, the contract can call `Permit2.transferFrom` function to transfer the tokens.

You can find these functions in IAllowanceTransfer. You will see there are 2 functions named `permit` and 2 named `transferFrom`. The reason there are 2 of each is the same as before – one is used for a single token transfer/approval and one for bulk. We will be using the single ones for this example.

```
function permit(address owner, PermitSingle memory permitSingle, bytes calldata signature) external;
```

`permit` is used to set the allowance, and it requires the following parameters:

- `owner` – token owner
- `permitSingle` – the `PermitSingle` message that includes the following data:
    - `details` – `PermitDetails` data that includes:
        - token address
        - amount to allow
        - expiration of allowance
        - nonce
    - `spender` – address that is being allowed to spend the tokens
    - `sigDeadline` – when the permit signature expires
- `signature` – signed permit message hash by the `owner`

```
function transferFrom(address from, address to, uint160 amount, address token) external;
```

`transferFrom` is used to transfer the tokens, and the parameters here should be self explanatory.

To be able to call these function we will need IAllowanceTransfer interface that has everything needed for allowance based transfers. Copy the interface to your codebase and import it. (I have put this one in `interfaces` directory as well.)

```
import {IAllowanceTransfer} from "./interfaces/IAllowanceTransfer.sol";
```

Now let's recreate the `deposit` function, so it accepts a permit to set an allowance and transfer the tokens.

```
function deposit(
    uint160 _amount,
    address _token,
    IAllowanceTransfer.PermitSingle calldata _permit,
    bytes calldata _signature
  ) external {
    tokenBalancesByUser[msg.sender][_token] += _amount;

    // 1. Set allowance using permit
    PERMIT2.permit(
      // Owner of the tokens and signer of the message.
      msg.sender,
      // The permit message.
      _permit,
      // The packed signature that was the result of signing
      // the EIP712 hash of `_permit`.
      _signature
    );

    // 2. Transfer the tokens
    PERMIT2.transferFrom(
      msg.sender,
      address(this),
      _amount,
      _token
    );
  }
```

Notice that the user can set allowance higher than the amount that is being transferred. This means that in the first deposit the allowance can be set high enough that it would not be needed to `permit` again when depositing the second time. Also, validating the signature and setting the allowance on every deposit might not be desired, because it means more gas will be used for the transaction. For that reason you can create another `deposit` function that calls only `transferFrom`.

```
function deposit(
    uint160 _amount
    address _token,
  ) external {
    tokenBalancesByUser[msg.sender][_token] += _amount;

    PERMIT2.transferFrom(
      msg.sender,
      address(this),
      _amount,
      _token
    );
  }
```

Now when user interacts with the contract for the first time, he can call the `deposit` function with permit to set the allowance and transfer the tokens, and next time he just calls the function without permit.

How to sign a permit

We covered the implementation on contract side, now all we need to do is figure out how to interact with it from the front end. We will be using NodeJS and Ethers. This guide assumes that you already set up an NodeJS app, imported ethers and connected a wallet.

For this example, we will try to deposit tokens using the function we created earlier that uses `permitWitnessTransferFrom`.

First add `@uinswap/permit2-sdk` package, since it will make our job much easier.

```
npm install @uniswap/permit2-sdk
```

There are a few things we will import, that will help us.

```
import {
  // permit2 contract address
  PERMIT2_ADDRESS,
  // the type of permit that we need to sign
  PermitTransferFrom,
  // Witness type
  Witness,
  // this will help us get domain, types and values that we need to create a signature
  SignatureTransfer
} from "@uniswap/permit2-sdk";
```

Before we can start depositing any tokens, we need to approve the `Permit2` contract to transfer our ERC20 tokens.

```
const erc20 = new ethers.Contract(erc20Address, erc20Abi, signerOrProvider);
await erc20.approve(PERMIT2_ADDRESS, constants.MaxUint256);
```

Tip: You can get `constants` from `ethers` or `@uniswap/permit2-sdk`.

Create a Contract instance for the contract that you want to interact with.

```
const contract = new ethers.Contract(contractAddress, contractAbi, signerOrProvider);
```

Create a permit.

```
const permit: PermitTransferFrom = {
  permitted: {
    // token we are permitting to be transferred
    token: erc20.address,
    // amount we are permitting to be transferred
    amount: amount
  },
  // who can transfer the tokens
  spender: contract.address,
  nonce: 1,
  // signature deadline
  deadline: constants.MaxUint256
};
```

Create a witness.

```
const witness: Witness = {
  // type name that matches the struct that we created in contract
  witnessTypeName: 'Witness',
  // type structure that matches the struct
  witnessType: { Witness: [{ name: 'user', type: 'address' }] },
  // the value of the witness.
  // USER_ADDRESS is the address that we want to give the tokens to
  witness: { user: USER_ADDRESS },
}
```

Next, we will sign the permit message. To sign a message using `_signTypedDat`a we need `domain`, `types` and `values`. `Domain` is the EIP-712 domain separator. In our case it includes `verifyingContract` address, which is `Permit2`, `chainId`, which is the chain ID where the contract that we are interacting with is deployed, and `name` of the contract. `Types` describe the structure of the message data and `values` are of course the values that we want to sign. Doing this by ourselves can be a bit tricky, but luckily the permit2 SDK has our back and all we need is to write these two lines to create the signature:

```
const { domain, types, values } = SignatureTransfer.getPermitData(permit, PERMIT2_ADDRESS, CHAIN_ID, witness);
let signature = await wallet._signTypedData(domain, types, values);
```

Now we can deposit the token.

```
await contract.deposit(amount, erc20.address, wallet.address, USER_ADDRESS, permit, signature);
```

And this is it. If you are interested in full code or examples of bulk transfers and permits, you can check my github repo.