Solidity Timeout

A decentralized marketplace for executing transactions in future, using GraphProtocol

CURRENTLY HOSTED ON RINKEBY

Problem

On solidity, we cannot schedule a transaction to be automatically executed at a future date.

For example, if a withdrawal from a Plasma Chain should be executed after 7 days, the transaction cannot be scheduled automatically. The user who is trying to make the withdrawal must execute a transaction after

7 days to claim the money into their wallet.

But, what if - the contract for withdrawal sets a transaction to be executed 7 days from today and the transaction is automatically executed at a later date. The user doesn't have to manually come and do one additional transaction. A major improvement in user experience.

Solution

Super simple.

There is a deployed contract called Timeout

. Derives its name from window.setTimeout

function in js.

Using this contract, a contract can automatically set a transaction to be executed at a later date by calling TimeoutContract.enqueue

.

Parameters to this function are

- contract address

: The contract at which the desired function resides. Type address

- function signature

: The signature of the function to be called. E.g. "callback(bytes)"

. Type string

. The function should accept a bytes

parameter

- start block

and end block

: The duration in which the function should be called. This is a range instead of a particular block because we cannot guarantee the block in which the transaction will be confirmed.

- value

: The enqueue

function is of type payable

. Use of this described below.

For example,

function exampleScheduler() public payable returns(uint) {

```
    ITimeout mTimeout = ITimeout("0xc6618ff525Dc8a054957a8650a54f107B82996F9");
    uint id = mTimeout.enqueue{ value: msg.value }(address(this), "callback(bytes)", abi.encode(0), block.number, block.number + 10);

    return id;

}
```

Timeout contract snippet

```
function enqueue(address contractAddress, string calldata signature, bytes calldata parameters, uint blockStart, uint blockEnd) public payable returns(uint){

    _id.increment();

    functions[_id.current()] = Function({

        contractAddress: contractAddress,

        signature: signature,

        parameters: parameters,

        blockStart: blockStart,

        blockEnd: blockEnd,

        value: msg.value,

        status: Status.QUEUED

    });

    emit CallFunction(_id.current(), blockStart, blockEnd, msg.value);

    return _id.current();

}

function call(uint id) public{

    Function memory f = functions[id];

    require(block.number >= f.blockStart && block.number <= f.blockEnd, "Function call not in specified range");

    require(f.status == Status.QUEUED, "Not in queue");

    f.status = Status.RUNNING;

    functions[id] = f;

    address(f.contractAddress).call(

        abi.encodeWithSelector(

            bytes4(

                keccak256(bytes(f.signature))

            ),

            f.parameters

        )

    );

    payable(address(msg.sender)).transfer(f.value);

    f.status = Status.COMPLETED;

    functions[id] = f;

    emit Executed(id);

}
```

Try it out

- Timeout contract
- A sample contract to schedule events

Marketplace

All the enqued events are available for query using a TheGraph

graphql endpoint.

A user can run a script to query for any scheduled (aka enqueued) function calls that need to be executed in the current block.

If yes, the script would execute the transaction. The value

associated with the function call is the reward for this script to submit the transaction

Realistically, the script would execute the transaction only if the gas fees to execute the transaction is greater than the value

i.e. reward

Ask

This has been hacked together in a couple hours and not audited. I’m definitely not an expert in security and gas optimizations. If you find a flaw or something we can do better, please do submit a PR!

Particularly, I’m looking for ideas to prevent the front running caveat that this marketplace is exposed to.

Caveat

Though there is a reward for the script that is submitting the transaction. The transactions will almost always get frontrun by miners. Leading to the script’s transaction failing, and the miner getting the reward.

Can this be made front-run proof?

How can this be improved?

See also :Github Repo