Importing Dependencies

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

import "@aave/core-v3/contracts/interfaces/IPool.sol";

import "@uniswap/v2-periphery/contracts/interfaces/IUniswapV2Router02.sol";

* **IERC20.sol**: Allows interaction with ERC-20 tokens.
* **ReentrancyGuard.sol**: Prevents reentrancy attacks (a common security issue in Solidity).
* **IPool.sol**: Interface for AAVE’s lending pool (used for flash loans).
* **IUniswapV2Router02.sol**: Interface for Uniswap/PancakeSwap to swap tokens.

## **2. Declaring State Variables**

address public owner;

IPool public aaveLendingPool;

IUniswapV2Router02 public dex1;

IUniswapV2Router02 public dex2;

* **owner**: Stores the contract owner's address.
* **aaveLendingPool**: Reference to AAVE’s lending pool for flash loans.
* **dex1 & dex2**: References to two decentralized exchanges (DEXs) used for arbitrage.

## **3. Constructor (Initialization)**

constructor(address \_aaveLendingPool, address \_dex1, address \_dex2) {

owner = msg.sender;

aaveLendingPool = IPool(\_aaveLendingPool);

dex1 = IUniswapV2Router02(\_dex1);

dex2 = IUniswapV2Router02(\_dex2);

}

* The constructor sets:
  + The contract deployer as the owner.
  + The aaveLendingPool, dex1, and dex2 addresses.
* \_dex1 and \_dex2 should be set as Uniswap, PancakeSwap, or other compatible DEXs.

## **4. Owner-Only Access Modifier**

modifier onlyOwner() {

require(msg.sender == owner, "Not the owner");

\_;

}

* Restricts function access to only the contract **owner**.

## **5. Initiating the Flash Loan**

function executeArbitrage(

address token,

uint amount,

address[] memory path1,

address[] memory path2

) external onlyOwner nonReentrant {

aaveLendingPool.flashLoanSimple(address(this), token, amount, "", 0);

}

* **Parameters**:
  + token: The token to borrow (e.g., USDC, ETH).
  + amount: The loan amount.
  + path1: Swap path for the first trade.
  + path2: Swap path for the second trade.
* Calls **AAVE’s flash loan function**, borrowing amount of token.

## **6. Executing Arbitrage (Triggered by AAVE)**

function executeOperation(

address asset,

uint amount,

uint premium,

address initiator,

bytes calldata params

) external returns (bool) {

**Triggered automatically by AAVE** after the loan is issued.

**Parameters**:

* asset: Token borrowed.
* amount: Borrowed amount.
* premium: Flash loan fee.
* initiator: Who requested the flash loan.
* params: Additional execution data (unused here).

## **7. Swapping Tokens for Arbitrage**

uint balanceBefore = IERC20(asset).balanceOf(address(this));

* Stores the contract's token balance before execution.

### **First Swap (DEX1)**

IERC20(asset).approve(address(dex1), amount);

uint[] memory amounts1 = dex1.swapExactTokensForTokens(

amount, 0, path1, address(this), block.timestamp + 60

);

Approves dex1 (Uniswap/PancakeSwap) to spend tokens.

Swaps amount for another token.

Stores the received amount in amounts1.

uint amountReceived = amounts1[amounts1.length - 1];

IERC20(path1[path1.length - 1]).approve(address(dex2), amountReceived);

uint[] memory amounts2 = dex2.swapExactTokensForTokens(

amountReceived, 0, path2, address(this), block.timestamp + 60

);

* Takes the swapped token from dex1 and swaps it on dex2.
* Converts back to the original token for profit.

## **8. Checking Profitability**

solidity

uint finalAmount = amounts2[amounts2.length - 1];

require(finalAmount > amount + premium, "No profit");

* Ensures the arbitrage trade **makes a profit** after repaying the flash loan.

## **9. Repaying the Flash Loan**

IERC20(asset).transfer(msg.sender, amount + premium);

return true;

* Repays the flash loan (amount + premium).
* Any extra tokens left are the arbitrage **profit**.

### **🛠️ Next Steps**

1. **Deploy the contract** on Ethereum or Binance Smart Chain.
2. **Monitor DEX prices** using an off-chain script (Python bot).
3. **Trigger executeArbitrage** when price differences are found.

### **Python Arbitrage Scanner & Executor**

This script will:

1. **Fetch token prices** from Uniswap and PancakeSwap.
2. **Detect arbitrage opportunities** based on price differences.
3. **Trigger the Solidity contract** to execute trades.

from web3 import Web3

import json

import time

# Connect to Ethereum/BSC network

INFURA\_URL = "https://mainnet.infura.io/v3/YOUR\_INFURA\_KEY"

web3 = Web3(Web3.HTTPProvider(INFURA\_URL))

# Contract details

CONTRACT\_ADDRESS = "YOUR\_SMART\_CONTRACT\_ADDRESS"

OWNER\_ADDRESS = "YOUR\_WALLET\_ADDRESS"

PRIVATE\_KEY = "YOUR\_WALLET\_PRIVATE\_KEY"

# Uniswap & PancakeSwap Router addresses

UNISWAP\_ROUTER = "0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f"

PANCAKESWAP\_ROUTER = "0xBCfCcbde45cE874adCB698cC183deBcF17952812"

# Load contract ABI

with open("FlashLoanArbitrage.json") as f:

contract\_abi = json.load(f)

contract = web3.eth.contract(address=CONTRACT\_ADDRESS, abi=contract\_abi)

# Token details (modify as needed)

TOKEN\_ADDRESS = "0xTOKEN\_CONTRACT\_ADDRESS"

AMOUNT = web3.toWei(1, 'ether') # 1 token

def get\_price(dex\_router, token):

"""Fetch token price from Uniswap or PancakeSwap"""

router = web3.eth.contract(address=dex\_router, abi=contract\_abi)

path = [token, router.functions.WETH().call()]

amounts = router.functions.getAmountsOut(AMOUNT, path).call()

return amounts[1]

def check\_arbitrage():

"""Detect arbitrage opportunity and execute trade if profitable"""

price\_uniswap = get\_price(UNISWAP\_ROUTER, TOKEN\_ADDRESS)

price\_pancakeswap = get\_price(PANCAKESWAP\_ROUTER, TOKEN\_ADDRESS)

if price\_uniswap < price\_pancakeswap:

print("Arbitrage opportunity found! Executing trade...")

execute\_trade(TOKEN\_ADDRESS, AMOUNT)

else:

print("No arbitrage opportunity found.")

def execute\_trade(token, amount):

"""Trigger Solidity contract to perform arbitrage"""

nonce = web3.eth.getTransactionCount(OWNER\_ADDRESS)

txn = contract.functions.executeArbitrage(

token, amount,

[UNISWAP\_ROUTER, PANCAKESWAP\_ROUTER],

[PANCAKESWAP\_ROUTER, UNISWAP\_ROUTER]

).buildTransaction({

'gas': 500000,

'gasPrice': web3.toWei('5', 'gwei'),

'from': OWNER\_ADDRESS,

'nonce': nonce

})

signed\_txn = web3.eth.account.signTransaction(txn, PRIVATE\_KEY)

tx\_hash = web3.eth.sendRawTransaction(signed\_txn.rawTransaction)

print(f"Transaction sent: {web3.toHex(tx\_hash)}")

# Run bot every 10 seconds

while True:

check\_arbitrage()

time.sleep(10)

### **How It Works**

1. **Fetches token prices** from Uniswap & PancakeSwap.
2. **Compares prices** to find an arbitrage opportunity.
3. **Calls the Solidity contract** to execute a flash loan trade.
4. **Repeats every 10 seconds** to scan for new opportunities.

### **🔹 Next Steps**

* Replace **YOUR\_INFURA\_KEY**, **YOUR\_WALLET\_ADDRESS**, and **YOUR\_SMART\_CONTRACT\_ADDRESS**.
* Deploy on **Ethereum** or **BSC**.
* Run on a VPS or local machine **24/7**.

This is the solidity code entirely

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

import "@aave/core-v3/contracts/interfaces/IPool.sol";

import "@uniswap/v2-periphery/contracts/interfaces/IUniswapV2Router02.sol";

/\*\*

\* @title FlashLoanArbitrage

\* @dev This contract executes arbitrage trades using flash loans from AAVE

\* and swapping assets between two decentralized exchanges (DEXs).

\*/

contract FlashLoanArbitrage is ReentrancyGuard {

address public owner; // Address of the contract owner

IPool public aaveLendingPool; // AAVE lending pool interface

IUniswapV2Router02 public dex1; // First DEX (e.g., Uniswap)

IUniswapV2Router02 public dex2; // Second DEX (e.g., PancakeSwap)

/\*\*

\* @dev Sets the contract owner and initializes the lending pool and DEX addresses.

\* @param \_aaveLendingPool Address of the AAVE lending pool

\* @param \_dex1 Address of the first DEX router (e.g., Uniswap)

\* @param \_dex2 Address of the second DEX router (e.g., PancakeSwap)

\*/

constructor(address \_aaveLendingPool, address \_dex1, address \_dex2) {

owner = msg.sender;

aaveLendingPool = IPool(\_aaveLendingPool);

dex1 = IUniswapV2Router02(\_dex1);

dex2 = IUniswapV2Router02(\_dex2);

}

/\*\*

\* @dev Restricts function access to only the contract owner.

\*/

modifier onlyOwner() {

require(msg.sender == owner, "Not the owner");

\_;

}

/\*\*

\* @dev Initiates a flash loan for arbitrage trading.

\* @param token Address of the token to borrow

\* @param amount Amount of tokens to borrow

\* @param path1 Swap path for the first DEX trade

\* @param path2 Swap path for the second DEX trade

\*/

function executeArbitrage(

address token,

uint amount,

address[] memory path1,

address[] memory path2

) external onlyOwner nonReentrant {

aaveLendingPool.flashLoanSimple(address(this), token, amount, "", 0);

}

/\*\*

\* @dev This function is called by AAVE after the flash loan is issued.

\* It executes arbitrage trades between two DEXs.

\* @param asset Address of the borrowed asset

\* @param amount Amount of borrowed asset

\* @param premium Flash loan fee

\* @param initiator Address that initiated the loan

\* @param params Additional parameters (unused here)

\* @return Returns true if successful

\*/

function executeOperation(

address asset,

uint amount,

uint premium,

address initiator,

bytes calldata params

) external returns (bool) {

require(msg.sender == address(aaveLendingPool), "Unauthorized");

uint balanceBefore = IERC20(asset).balanceOf(address(this));

// Swap borrowed asset on dex1

IERC20(asset).approve(address(dex1), amount);

uint[] memory amounts1 = dex1.swapExactTokensForTokens(

amount, 0, path1, address(this), block.timestamp + 60

);

uint amountReceived = amounts1[amounts1.length - 1];

// Swap received asset on dex2

IERC20(path1[path1.length - 1]).approve(address(dex2), amountReceived);

uint[] memory amounts2 = dex2.swapExactTokensForTokens(

amountReceived, 0, path2, address(this), block.timestamp + 60

);

uint finalAmount = amounts2[amounts2.length - 1];

require(finalAmount > amount + premium, "No profit"); // Ensure profit is made

// Repay flash loan

IERC20(asset).transfer(msg.sender, amount + premium);

return true;

}

}

To **deploy and run** the Flash Loan Arbitrage contract, follow these steps:

## **1️⃣ Install Dependencies**

You'll need **Node.js**, **Hardhat**, and dependencies like OpenZeppelin and AAVE.

# Install Hardhat

npm install -g hardhat

# Create a new Hardhat project

mkdir flash-loan-bot && cd flash-loan-bot

npx hardhat

# Install required dependencies

npm install @openzeppelin/contracts @aave/core-v3 @uniswap/v2-periphery dotenv

## **Configure Hardhat for Ethereum or BSC**

Edit hardhat.config.js and add your **network RPC URLs** (Infura for Ethereum, BSC RPC for Binance Smart Chain).

require("@nomicfoundation/hardhat-toolbox");

require("dotenv").config();

module.exports = {

solidity: "0.8.20",

networks: {

ethereum: {

url: process.env.ETHEREUM\_RPC\_URL,

accounts: [process.env.PRIVATE\_KEY],

},

bsc: {

url: process.env.BSC\_RPC\_URL,

accounts: [process.env.PRIVATE\_KEY],

},

},

};

Create a .env file and add:

ETHEREUM\_RPC\_URL="https://mainnet.infura.io/v3/YOUR\_INFURA\_KEY"

BSC\_RPC\_URL="https://bsc-dataseed.binance.org/"

PRIVATE\_KEY="YOUR\_WALLET\_PRIVATE\_KEY"

## **Deploy the Contract**

Create a scripts/deploy.js script:

const hre = require("hardhat");

async function main() {

const FlashLoanArbitrage = await hre.ethers.getContractFactory("FlashLoanArbitrage");

const contract = await FlashLoanArbitrage.deploy(

"AAVE\_LENDING\_POOL\_ADDRESS",

"UNISWAP\_ROUTER\_ADDRESS",

"PANCAKESWAP\_ROUTER\_ADDRESS"

);

await contract.deployed();

console.log(`Contract deployed at: ${contract.address}`);

}

main().catch((error) => {

console.error(error);

process.exit(1);

});

Run deployment:

npx hardhat run scripts/deploy.js --network ethereum

## **Execute the Arbitrage**

You can now interact with the contract using **Hardhat scripts** or a **Python bot**.