Documentation for Blockchain Programming project:

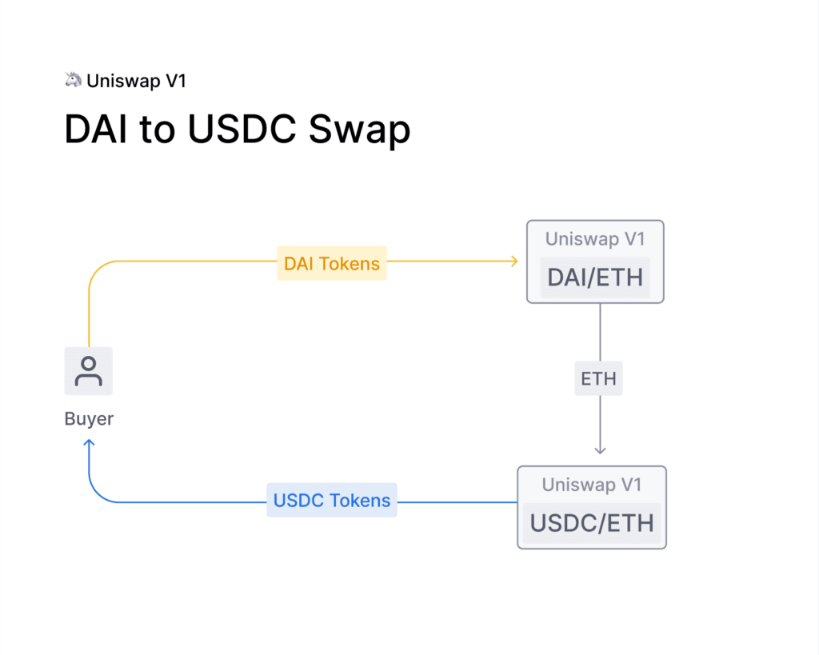
Goal: The goal is too able to implement decentralized exchanges such as Uniswap in our university network and to use it with contracts to perform certain specialized actions. We are trying to understand what a flash loan attack is in code, how to implement it successfully in contracts, and use chain-link price feeds to see how much can be taken and that it is taken by the flash loan attack.

Significance: Currently, this a very important issue as many blockchain networks is being subject to these kinds of attacks and are losing millions. A flash loan attack occurs when the borrower uses the markets as the loan is taking place, driving the value of the stolen token underwater thanks to additional slippage, and then allowing the attacker to buy back the token at a depressed amount. These flash loan attacks take benefit of the leverage provided by flash loans to allow an attacker to develop weaknesses within DeFi projects' smart contracts. In many cases, these exploits allow the attacker to totally drain a project's liquidity pools, racking up massive losses for the protocols' clients. Conventional lenders take on two types of risk. The initial one is default risk: if the borrower runs off with the money, that clearly is terrible. But the second risk to a lender is the illiquidity risk: if a lender lends out too many of its assets at the wrong times or doesn’t obtain judicious repayments, the lender may be suddenly illiquid and not be able to meet its own commitments. In other words, a flash loan functions as the following “I will lend you as much money as you want for this one transaction. But by the close of this transaction, you must pay me at the slightest as much as I lent you. If you are incapable to do that, I will certainly roll back your contract”.

In addition, there are major security issues in blockchain transactions, which makes the flash loan attack and its varying types so attractive to cybercriminals. All flash attacks should eventually be obtained by miners. This will serve as a warning against flash attacks since it will leave attackers powerless to mould their discoveries of these vulnerabilities. Flash loans are used non-spitefully to take advantage of arbitrage prospects across various exchanges. Flash loans have been increasingly used in attacks on DeFi protocols such as with Cheese bank and Harvest.

Programming and contracts: We are using UZHETH network smart contracts, and the programming being done is Solidity with these contracts and running simulated flash loan attack code and testing the amount being taken with the Metamask extension with input liquidity.

Uniswap: Decentralized exchanges are not in any decentralized oracles. Using Uniswap, Sushiswap, or Curve to get pricing information to execute trades is dragging data from protocols whose price depends solely on liquidity. Looking at the infamous ground zero bZx attack that sparked this wave of attacks is very important. The issue here relies in the fact that these protocols prices vary completely on liquidity. First, the user takes a massive loan out, uses that loan, to buy out all the liquidity on one side of a liquidity pool, causing a massive drop in price, and lastly this price is being used by another protocol to determine or execute some code such as they peg the price of their asset to the protocol’s asset. So, the user swaps for next to nothing for a huge profit. Then pays back the original loan and goes with these huge gains.

What auditors and software engineers need to do, is make sure they don't get valuing or data that rely on DEXes. Uniswap is a decentralized exchange NOT a decentralized pricing oracle, they are each a centralized data point, and using them is serious to a point where in the last 2 months, about 5 protocols have been hacked for over thirty million dollars combined.

Comparison on Uniswap versions:

The Uniswap in the diagram above is based on the Automated Market Maker (AMM) model. This model trusts on a mathematical formula to price holdings. Rather than placing orders, AMMs rely on Liquidity Providers (LPs) who spend trading pairs in liquidity pools. Uniswap is a Constant Function Market Maker, which means that the proportion of trading pairs in every liquidity pool must respect the Constant Product Formula:

**x\*y=k**

k is a constant, x is the reserve of the first asset, and y is the reserve of the second asset. This means that all the liquidity pools are to provide additional liquidity so that it wouldn’t change k. Also, everyone traded had to be aware of the total amount of the funds locked to prevent high slippage. Uniswap v1 provides for only ETH-ERC20 trading pairs, so you could only swap ETH for a single ERC20 token. So, if you wanted to swap USDC for DAI, you had to swap USDC for ETH and then go to the ETH-DAI pool to get DAI.

Then, the Uniswap v2 was a much safer and more user-friendly version of Uniswap v1. The main problem of v1 adopted in this new version was the absence of ERC20-ERC20 token pools. This suffered much higher costs and slippage for users who wanted to exchange one ERC20 token for another. Uniswap v2 also applies a new functionality that enables highly decentralized and manipulation-resistant on-chain price feeds. For this version, we must calculate the average price over a period of blocks (Time Weighted Average Price) by dividing the cumulative price (sum of the Uniswap price in the entire history of the contract) by the timestamp duration (the end-of-duration minus the start-of-duration timestamp).

Ethereum network: Ethereum has the Defi protocol unlike Bitcoin or Polkadot which makes it ideal for processing smart contracts with the flash loan attack code and the solution of the chain link price feeds using the Metamask chrome extension.

Chain link price feed: Chain-link Data Feeds are the fastest way to link your smart contracts to the real-world market prices of assets. For example, one purpose for data feeds is to allow smart contracts to recover the latest pricing data of an asset in a single call. Data feeds are available on networks such as EVM-compatible networks, Solana, and Terra. The solution is it needs to come from decentralized oracles and get the data from decentralized Chain-link Price Feeds if it's price data. For any other data you need to get your information from a decentralized network of Chain-link Oracles. Anyone can modify their oracle network to make it as broad or thin as they like.

This all being said, at this point, there is enough data out there that if a protocol gets hacked and that protocol reimbursed an auditor, that auditor needs to be held responsible as well, as lost centralized price oracles in audit reviews is going to make this keep occurring. Many projects who have been hacked have combined Chain-link price feeds as their support for data reliability.

Prevention: Use the built-in transfer () function. It only sends 2300 gas with the external call, making reentrancy almost difficult. Since that will give you just enough gas to write to a log.

You could instead add a mutex, or a variable that puts a lock on calling the function or working with the variables until the work with them is done. You don't need to do all these tips, but you do need to do at least one of these tips.

**Prevention Example**

This code:

1function withdraw() external {

2 uint256 amount = balances[msg.sender];

3 require(token.transfer(msg.sender, amount)());

4 balances[msg.sender] = 0;

5}

Should be changed so that the external token transfer call occurs **after** the balance is revised to 0.

1function withdraw() external {

2 uint256 amount = balances[msg.sender];

3 balances[msg.sender] = 0;

4 require(token.transfer(msg.sender, amount)());

5}

Also, you could do this:

1bool public mutex = false;

2

3function withdraw() external {

4 require(!mutex);

5 mutex = true;

6 uint256 amount = balances[msg.sender];

7 balances[msg.sender] = 0;

8 require(token.transfer(msg.sender, amount)());

9 mutex = false;

MetaMask Chrome extension: MetaMask is a wallet that functions as a self-controlled application inside your browser as an expansion. It permits you to relate with Decentralized Applications or "dApps". MetaMask makes retrieving the Ethereum & Testnet blockchains (Like Kovan) very easy and offers you with wallet addresses.

Video:

The video will show the changing token balances in the smart contracts. As a result of our flash loan attack and our work with Uniswap and Sushiswap, the amount of ETH and the amount of tokens to return to get the price on exchange can change.

Errors during process and solutions:

1. UNISWAPV2ERC20 – 0x057f2215ce92Df45262E87bd65E81646F8f6BE18

UNISWAPV2FACTORY – 0xa2AE2e5339E26cf76a610E604373a4E574AE59c2

UNISWAPV2PAIR – 0x249Be9ABeC4a9C92373D3CA62BF182f67B1F22f1

UNISWAPV2ROUTER02 – 0xF2D796EFE2745efb2b7B78A64D52A528CE55e62c

WETH – 0xbD29eC7f0De30cF0be9b5d9782d251aFEEcfecEA

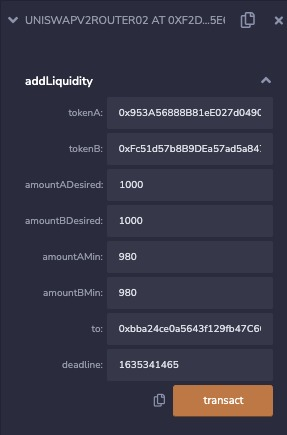
UZHUSDT – 0x953A56888B81eE027d049011D65785306138F22c

UZHDAI – 0xFc51d57b8B9DEa57ad5a84700Be15d9C3a651E9F

We successfully created two tokens (UZHUSDT, UZHDAI) and paired them on uniswap via the UNISWAPV2FACTORY contract. Furthermore, we added these tokens to Pascal's Metamask (see attachment)

However, we are struggling to create our very first Liquidity Pool. To be more precise, we wanted to call the addLiquidity Function in UNISWAPV2ROUTER02 (see attachment) We tried to create a UZHDAI/UZHUSDT Pool, but we receive the following error message:

Internal JSON-RPC error. {"code": -32000, "message": "execution reverted" }¨



1. The onFlashLoan() on the FlashBorrower is executed after the FlashBorrower has received the flash loan from the FlashLender. We have adapted its content to the steps of a flash loan attack. Plus, we added “IERC20(token). transfer (msg.sender, IERC20(token).balanceOf(address(this)));” at the end of the flashBorrow() on FlashBorrower to finally send the exploited tokens to the attacker.

We got an error in flashBorrow() on the FlashBorrower because of “IERC20(token).approve(address(lender), \_allowance + \_repayment);”. At this point the FlashBorrower doesn’t have \_allowance + \_repayment of the token and therefore can’t approve it to the FlashLender. We had to move this part to onFlashLoan() on the FlashBorrower because only at this point the FlashBorrower has received the flash loan.

The new addresses were:

FLASHLENDER – 0x4e99557ac0b9E038A325c8e8b42cc367d2B1e2e8

FLASHBORROWER – 0x6f58D94785762e84d123DcB36b85fD8EFe58f96a

1. We took loan.sol as template for our newly created CollateralLoan contract. We struggled with getting the current token prices from uniswap, so we did that manually within CollateralLoan according to its liquidity pool reserves.

The address we used:

COLLATERALLOAN – 0xB67589Dca97441371c5bf360eb8c18Cb35621c48

1. Preconditions

- \_isFlashLoanAttackPossible is true on CollateralLoan

- setup a uniswap pool with low liquidity which we want to manipulate during the flash loan attack

- FlashLender must have enough tokens to lend

- CollateralLoan must have enough tokens to lend

Execution

Call flashBorrow() on FlashBorrower and check your wallet afterwards

1. For the chainlink connection to the smart contracts suffering with and without the flash loan attack we have two options. The first is the local deployment using a chain link node and the second is to use a constructor and an aggregatorV3interface to have it linked with a testnet using a contract address. Both options can work, but for simplicity and for the sake of less errors we will use the local deployment chainlink node option.

Solution to issue: Add liquidity and changed the `PERMIT\_TYPEHASH` variable in v2-periphery and `INIT\_CODE\_HASH` variable in v2-sdk for the deployment (as in the articles above).

1. We are having issues in deploying the flash loan in the UZHETH network. First, we must implement a flash loan provider (such as Aave, AVAX etc.) and second, we must integrate a chain link oracle.

However, we thought that maybe there are already deployed smart contracts for conducting a flash loan on the UZHETH? In the hopes that we could focus on integrating the chainlink. We found a relatively simple documentation on how to deploy an AAVE application in remix. Unfortunately, these documentations always use Ropsten or another Ethereum Testnet which does not really help.

1. We fixed the error from last time (Internal JSON-RPC error. {"code": -32000, "message": "execution reverted”} ¨) by changing the INIT\_CODE\_PAIR\_HASH in UniswapV2Library.sol to the INIT\_CODE\_PAIR\_HASH of UNISWAPV2FACTORY.

We tried to add liquidity again. Then we got a new error: Internal JSON-RPC error. {"code": 3, "message": "execution reverted: TransferHelper:transferFrom: transferFrom failed"}. We solved it by first allow the UNISWAPV2ROUTER02 to access some of our tokens (by executing the approve function on both tokens)

We tried it again. Another error occurred: Internal JSON-RPC error. {“code”: 3, “message”: “execution reverted: ds-math-sub-underflow”}. We found out that we can only use tokens with 18 decimals. So, we deployed UZHSHIB2 and UZHUSDC2 with 18 decimals instead of 2. After trying all this it worked.

1. We are currently trying to implement the chain link solution to our smart contracts either through remix, local deployment by hard coding the values, and through Faucet test net solutions.

Sources used:

* <https://insights.glassnode.com/defi-attacks-flash-loans-centralized-price-oracles/>
* <https://docs.chain.link/docs/get-the-latest-price/>
* <https://data.chain.link/>
* <https://preventflashloanattacks.com/>
* <https://solidity-by-example.org/0.6/>
* For the deployment of the Uniswap Factory: <https://github.com/Uniswap/v2-core>
* Docs for Uniswap Factory: <https://docs.uniswap.org/protocol/V2/reference/smart-contracts/factory>
* For the deployment of the Uniswap Router: <https://github.com/Uniswap/v2-periphery>
* Docs for Uniswap Router: <https://docs.uniswap.org/protocol/V2/reference/smart-contracts/router-02>
* Uniswap deployment tutorial: <https://www.youtube.com/watch?v=U3fTTqHy7F4>
* Source code for the uniswap deployment tutorial: <https://github.com/jklepatch/eattheblocks/tree/master/screencast/229-fork-uniswap>
* <https://mvpworkshop.co/blog/uniswap-v3-explained-all-you-need-to-know/#Uniswap_v1_vs_v2_-_A_Brief_Introduction_and_Comparison>
* <https://blockchain.news/wiki/how-to-build-an-uniswap-exchange>
* <https://vomtom.at/how-to-use-uniswap-v2-as-a-developer/>
* <https://github.com/t4sk/defi-by-example/blob/main/contracts/TestUniswapLiquidity.sol>
* <https://docs.uniswap.org/protocol/V2/reference/smart-contracts/router-02#addliquidity>
* <https://market.link/nodes/eb5c92a8-6093-4657-9a68-a6d10719946e/integrations?network=1>
* <https://eattheblocks.com/how-to-perform-custom-ethereum-flash-loans-using-solidity-erc-3156-standard/>
* <https://programtheblockchain.com/posts/2018/03/06/writing-a-collateralized-loan-contract/>
* <https://ethereum.stackexchange.com/questions/91441/how-can-you-get-the-price-of-token-on-uniswap-using-solidity/94173>
* <https://medium.com/meter-io/the-bzx-attacks-what-went-wrong-and-the-role-oracles-played-in-the-exploits-264619b9597d>

Part 1 – Flash Loan Attack Success

**Precondition**

1. 2 Tokens

2. liquidity pool on uniswap (addLiquidity() on UniswapV2Router02) with it, UZHDOT (supply 1’000’000) & UZHUST (supply 1’000’000), liquidity pool with 40 UZHUST : 1 UZHDOT ratio, example 40000 UST : 1000 DOT) approve router address to claim tokens from your metamask to add it to the liquidity pool

Graphical user interface, application

Description automatically generatedGraphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generatedGraphical user interface

Description automatically generatedGraphical user interface, application

Description automatically generated

3. CollateralLoan has to be deployed and has to have enough tokenToBorrow (example: UZHDOT 500’000)

COLLATERALLOAN – 0x8661836E4BeEF2B398dE922580a994f6f3Ced062

Graphical user interface

Description automatically generated

**Text

Description automatically generated**

Note: the factory addresses of the exchanges in the CollateralLoan.sol smart contract are hardcoded because of simplicity

4. Flash Loan Possibility: FlashLender and FlashBorrower have to be deployed and FlashLender has to have enough tokensToFlashLoan (example: UZHDOT 99’000) an

deploy flashlender with UZHDOT as supported token

**Graphical user interface, application, Teams

Description automatically generated**

send enough tokens to the flash lender address (UZHDOT 99’000)

Graphical user interface, text

Description automatically generated

deploy FlashBorrower

FLASHBORROWER – 0xFCEb2EAcD90DE98692989285aA25255817F5bFC5

**Graphical user interface, application, Teams

Description automatically generated**

UniswapV2Router02 – 0x1ceae99b55aC4a79c0f74460E628067e30e38925

UZHUST – 0xc79A266B6A94461BeF93E668335274cCcC21C853

UZHDOT – 0x5CEC9849d71D090a65C310fF466eA8c45EaBd175

FLASHLENDER – 0x2e113573D35efD72659C36e7D9959A8A17a7E96e

**Action**

1. set isFlashLoanAttackPossible to true on CollateralLoanContract

Graphical user interface, website

Description automatically generated

2. check your balances and call flashLoanAttack on FlashBorrower accordingly (example: tokenToFlashLoan = UZHDOT, amountToFlashLoan = 10000, tokenToSwap = UZHUST)

Graphical user interface, application

Description automatically generated Graphical user interface, application, website

Description automatically generated

3. flashLoanAttack should be done, check your balances

Part 2 – Flash Loan Attack Prevention (average function)

Part 3 – Flash Loan Attack Prevention (chainlink oracle)

0x5CEC9849d71D090a65C310fF466eA8c45EaBd175

10000000000000000000000

0xc79A266B6A94461BeF93E668335274cCcC21C853