Flash Loan Attacks: Manipulating Liquidity Pools and options to prevent this kind of attacks.

Maximilian Kiefer (20-746-442)
Pascal Emmenegger (15-532-302)
Nicola Crimi (12-748-612)

Department of Informatics, University of Zurich

10.12.2021

ABSTRACT

We introduced a possibility to execute a Flash Loan Attack (FLA) on the UZHETH network and point out prevention methods. Our FLA is an oracle manipulation attack and exploits a collateralized loan provider (CLP) by first borrowing token A as flash loan which then will be swapped partly to token B on a decentralized exchange (DEX) and finally making profit by borrowing a loan of token A against token B from the CLP which calculates lending amounts according to the manipulated liquidity pool reserves. As prevention techniques, we suggest CLPs to use averaged token prices for lending amount calculations, respectively the introduction of price oracles, e.g. Chainlink. The code for our project is available at https://github.com/pemmenegger/Group033_Install_Uniswap_and_flash_loan_attack

Keywords: flash loan attack, oracle manipulation, liquidity pools, prevention

1 INTRODUCTION

Flash Loan Attacks (FLAs) are severe issues regarding the vulnerability of smart contracts within DeFi protocols as many blockchain networks are currently subject to these kinds of attacks and are losing millions of US dollars.

A FLA occurs when an individual uses the markets in its favours, i.e. manipulating price actions of DeFi protocols, while a flash loan is ongoing and profiting *before* the completion of the aforementioned loan. A popular method is driving the value of a token underwater and then allowing the attacker to buy back the token at a depressed amount. A FLA uses the possibility of flash loans to raise uncollateralized loans to attack DeFi protocols with the goal to benefit from possible weaknesses within its smart contracts.

In many cases, these exploits allow the attacker to significantly drain a project's liquidity pools, racking up massive losses for the protocol's clients. Conventional lenders take on two types of risk. The initial one is a default risk: if the borrower runs off with the money, that clearly is terrible. 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 does not obtain judicious repayments, the lender may be suddenly illiquid and not be able to meet its own commitments. In addition to these risks, there are also major security issues in smart contracts of DeFi protocols, which makes the FLA and its varying types so attractive to cybercriminals, thus adding another dimension to the vector of uncertainty for a lender.

Out of many different types of FLAs, we decided to conduct an oracle manipulation attack because it is the most performed FLA type on the Ethereum network so far. The intention of our FLA is to deceive a CLP's token value determination algorithm. First, the attacker takes a massive flash loan of token A out and uses that flash loan to buy out all the liquidity on one side of a liquidity pool which causes a massive drop in price. This price is being used by the token value determination algorithm of our CLP such that it pegs the price of its assets to the liquidity pool reserves ratio. Hence, the attacker can now borrow much more token

A against token B from the CLP that it should have been according to its real values and makes a huge profit. After repaying the flash loan, the attacker should have worked out a positive delta of token A which can be transferred to the corresponding hot wallet, e.g. Metamask.

As prevention technique for our FLA, we suggest CLPs to use averaged token prices by either manually access prices of multiple decentralized exchanges and computing its average or make use of Chainlink Oracles which already provide this functionality.

In summary, the contributions of this report are as follows.

1. We propose **a method to execute a FLA on UZHETH** which is based on the oracle manipulation technique. By providing the source code, deployment instructions and an execution guide it is reproducible.
2. We introduce **prevention techniques for our FLA** which rely on the notion of averaging token prices and are also repeatable by yourself.

2 CONCEPTUAL BACKGROUND

This section is dedicated to a general overview of the domain of decentralized exchanges, flash loans, FLAs and the recent history of FLAs and its fixes.

2.1 Decentralized Exchange (DEX)

A DEX is a cryptocurrency exchange which allows for direct peer-to-peer cryptocurrency transactions to take place without the need for an intermediary. In this project the focus is set on the price calculations of DEXs. Normally, decentralized exchanges do not obtain price information via decentralized oracles. Instead, popular exchanges like Uniswap, Sushiswap, or Curve get their price information via an internal price calculation method depending (almost) solely on their liquidity pools. This paved the way for the infamous ground zero bZx attack which launched a wave of similar attacks as shown in Table 1.

To successfully conduct a FLA, we are dependent on a functioning DEX on UZHETH and therefore decided to deploy Uniswap V2. Its source code is available on GitHub. In the following a short overview and comparison about Uniswap V1 and V2. Uniswap already released V3 in 2021 but considering Uniswap V3 for this project was out of scope.

Uniswap V1 introduced an 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:

$$x * y = k \qquad (1)$$

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 k always stays the same. Also, everyone trading 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. Consequently, 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 which is illustrated in Figure 1.

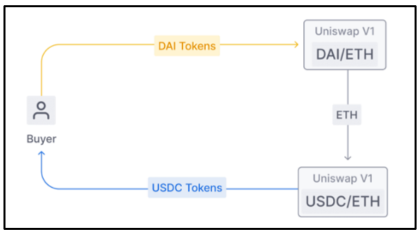


Figure 1 shows the procedure of swapping USDC for DAI

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. Hence, in V2 a USDC for DAI trade is executed as direct swap via the newly created Router and DAI/USDC liquidity pool which is depicted in Figure 2. However, these ERC20-ERC20 token pools suffered from much higher costs and slippage. 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).

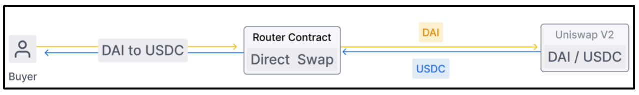


Figure 2 illustrates how a DAI/USDC swap works on Uniswap V2

2.2 Flash Loans

A flash loan is a relatively new possibility of uncollateralized lending offered by DeFi protocols. It is only valid within one blockchain transaction. Thus, flash loans fail if the borrower does not repay its debt before the end of the transaction. That is, because a blockchain transaction can be reverted during its execution. In other words, a flash loan functions as the following 'I will lend you the requested money for this one transaction if I own this amount. But by the close of this transaction, you must repay me as much as I lent you plus additional fees. If you are incapable to do so, I will roll back your contract'.

2.3 Flash Loan Attacks (FLA)

Since every FLA is slightly different than others, there is not a sharp line when it comes to a classification. Roughly, there are three different categories: pump & arbitrage, re-entrancy, and oracle manipulations.

Protocol	Value (in $)	Date	Attacking Type	Fix
bZx (1)	350’000	Feb. 2020	Pump & Arbitrage	?
bZx (2)	600’000	Feb. 2020	Oracle Manipulation	Chainlink Integration
Origin Protocol	7’000’000	Nov. 2020	Re-Entrancy	?
Harvest.Finance	24’000’000	Oct. 2020	Oracle Manipulation	?
Value Defi	6’000’000	Nov. 2020	Oracle Manipulation	Chainlink Integration
Akropolis	2’000’000	Nov. 2020	Re-Entrancy	Re-Entry Security
Cheese Bank	6’000’000	Nov. 2020	Oracle Manipulation	?
Compound	89’000’000	Nov. 2020	Oracle Manipulation	?
MakerDAO	Unknown	Nov. 2020	Oracle Manipulation	?
Warp Finance	7’760’000	Dec. 2020	Oracle Manipulation	?

Table 1 is an overview of the biggest FLAs in 2020

Pump & Arbitrage

The goal of a Pump & Arbitrage FLA is to make profit by using the flash loan of token A to swap it against token B on a DEX with a low valuation for token B and then trade this token B back to token A on another DEX with a higher valuation for token B. As the name of the attack reveals, this attack is making use of arbitrage trading.

Re-Entrancy

A re-entrancy attack can occur when you create a function that makes an external call to another untrusted contract before it resolves any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the effects were resolved.

Oracle Manipulations

These flash loans are used to crash and manipulate token prices on DEXs, which most projects deemed safe to use and rely on it. The issue here lies in the fact that these protocol’s token value determination mechanisms entirely depend on liquidity of one or a low amount of DEXs.

Since most of the conducted FLAs in 2020 were oracle manipulations, this report focuses as well on this type. Even within the category of oracle manipulations there is not a single pattern of attacks. On a high level the following steps are executed:

1. Taking out a massive loan, e.g. token A, from a protocol supporting flash loans.
2. Swapping token A for token B on a DEX, e.g. Uniswap, dumping the price of token A.
3. Deposit the purchased token B as collateral on a DeFi protocol that uses the above DEX as its sole price feed and borrow even more with this manipulated price.
4. Use a portion of borrowed token A to fully pay back the original flash loan and keep the remaining tokens.

2.4 Previous FLAs and its fixes

Since DeFi protocols allow flash loans from the end of 2019, many protocols suffered from all kinds of FLAs. Table 1 shows an overview about the biggest FLAs in 2020 and its fixes.

Our project focuses on the bZx (2) attack which was one of the first oracle manipulation attacks and has later been fixed by bZx with the help of a Chainlink integration.

3 PRECONDITIONS FOR THE FLA

In the following chapter we document the preconditions for successfully conducting an oracle manipulation. Every precondition consists of smart contracts which first must be deployed on the UZHETH network. In each case we explain the context and state how to deploy it. In Table 2 we reveal the addresses of our deployed smart contracts.

All the programming has been done within the remix IDE and in the Solidity language. The code can be found on GitHub. The browser extension Metamask is used as hot wallet. Thus, you need be familiar with the remix IDE in combination with Metamask to be able to follow and execute the deployment steps on the UZHETH network. Instead of deploying every precondition from scratch, you can also use our already deployed precondition smart contracts by adding it to your remix IDE.

Precondition	Smart Contract	Address
ERC20 Tokens	UZHDOT.sol	0x5CEC9849d71D090a65C310fF466eA8c45EaBd175
ERC20 Tokens	UZHUST.sol	0xc79A266B6A94461BeF93E668335274cCcC21C853
DEX: Uniswap	UniswapV2ERC20.sol	0x057f2215ce92Df45262E87bd65E81646F8f6BE18
DEX: Uniswap	UniswapV2Pair.sol	0x249Be9ABeC4a9C92373D3CA62BF182f67B1F22f1
DEX: Uniswap	UniswapV2Factory.sol	0xeB8749f7394AfE2A5cf44251d196763C1E2aC5Cb
DEX: Uniswap	WETH.sol	0xF85111AD8e5B6399a3CA9F88148D9373D7dB6ed0
DEX: Uniswap	UniswapV2Router02.sol	0x1ceae99b55aC4a79c0f74460E628067e30e38925
DEX: Sushiswap	SushiswapV2ERC20.sol	0x5D2ef1099daE80DA626d049203e76c6b59660415
DEX: Sushiswap	SushiswapV2Pair.sol	0xA2F0C3a838A78d88637a41f9bE160b283c37B641
DEX: Sushiswap	SushiswapV2Factory.sol	0xa04AFAf07bFb9E4E14a84fFC103490ABd7B9B927
DEX: Sushiswap	WETH.sol	0xCC941B54c0E9706548C084dDB0079539d92991Db
DEX: Sushiswap	SushiswapV2Router02.sol	0xaED9976746d7aC4db75D7D5F188D5606f762B446
CLP	CollateralLoan.sol	0x8661836E4BeEF2B398dE922580a994f6f3Ced062
Flash Loan Provider	FlashLender.sol	0x2e113573D35efD72659C36e7D9959A8A17a7E96e
Attacking Contract	FlashBorrower.sol	0xFCEb2EAcD90DE98692989285aA25255817F5bFC5

Table 2 is an overview of our deployed precondition smart contracts and its addresses

3.1 ERC20 Tokens

As we decided to build our FLA based on the oracle manipulation technique two tokens are needed. One is the token of which we want to borrow a flash loan and the other token is needed to create a liquidity pool and manipulate it.

For the deployment you can take *UZHUST.sol* and *UZHDOT.sol* as templates which can be adapted to your own tokens. Note that the tokens must have exactly 18 decimals for a successful FLA. Next, deploy your tokens with your desired supply. After successfully deploying the smart contracts and adding the token addresses to your Metamask you should see the total supply of these tokens in your wallet. For this project we minted two ERC20 tokens. More specifically, *UZHUST.sol* treated as token A and *UZHDOT.sol* as token B with a total supply of 1’000’000 each.

3.2 DEX with Liquidity Pool

To conduct a FLA within the UZHETH network it is also necessary to have at least one working DEX. As the UZHETH network does not have a DEX yet we decided to take Uniswap V2 (without GUI) and deploy it by ourselves. You can find the corresponding source code in *dependencies/uniswap-v2* of our GitHub repository which we forked from Uniswap’s publicly available repository.

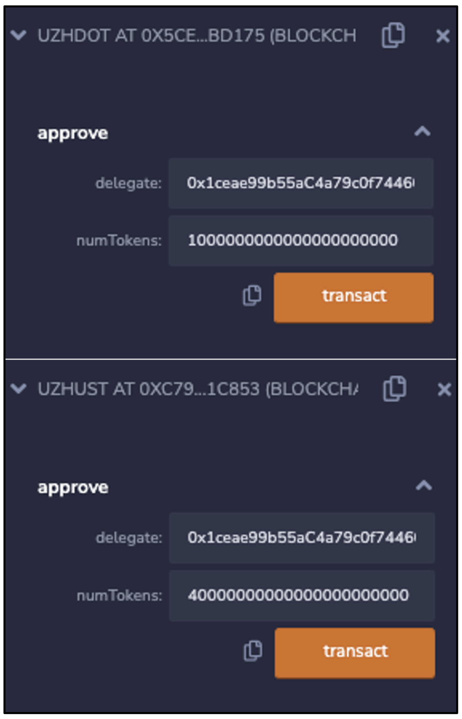
First, deploy *UniswapV2ERC20.sol, UniswapV2Pair.sol* and *WETH.sol*. Then, deploy *UniswapV2Factory.sol* with your Metamask wallet address as *_feeToSetter*. After the deployment went through request the *INIT_CODE_PAIR_HASH* attribute value on *UniswapV2Factory.sol*, remove the first two digits which should be ‘0x’ and copy it. Next, go to line 24 of the file *UniswapV2Library.sol* and paste it between the two quotation marks.

Finally, deploy *UniswapV2Router02.sol* with the addresses of *UniswapV2Factory.sol* as *_factory* and *WETH.sol* as *_WETH.*

The team decided on deploying Uniswap and Sushiswap. They both have been deployed with the same procedure as before described. The only difference is the naming of the source files. Uniswap is used for the FLA execution whereas Sushiswap is needed for the prevention of the FLA.

Furthermore, we now must initialize a liquidity pool with the two deployed tokens (UZHUST, UZHDOT).

As a first step approve the Uniswap Router to claim tokens from your Metamask wallet to add it to the liquidity pool. Therefore, execute the *approve* function with the address of *UniswapV2Router02.sol* as *delegate* and the token amount you want to add to the liquidity pool as *numTokens.* As you can see in Figure 3, we chose a ratio of 40 UZHUST to 1 UZHDOT and consequently want to add 40000 UZHUST and 1000 UZHDOT to the liquidity pool.



UZHDOT AT 0X5CE...BD175 (BLOCKCH

approve

delegate: 0x1ceae99b55aC4a79c0f7446

numTokens: 1000000000000000000000

transact

UZHUST AT 0XC79...1C853 (BLOCKCH

approve

delegate: 0x1ceae99b55aC4a79c0f7446

numTokens: 40000000000000000000000

transact

Figure 3 depicts an example for executing an approval for our two tokens

Finally, execute the *addLiquidity* function on *UniswapV2Router02.sol* to effectively add some of your tokens to its liquidity pool. According to Figure 4 we mapped *tokenA* to UZHUST and *tokenB* to UZHDOT. Thus, *tokenA*, *amountADesired* and *amountAMin* have been set to the address of UZHUST, 40000 UZHUST and 1 UZHUST whereas *tokenB*, *amountBDesired* and *amountBMin* have been filled with the address of UZHDOT, 1000 UZHDOT and 1 UZHDOT. The *to* attribute represents the address which should receive the liquidity tokens afterwards. In our case it is our Metamask wallet address. The *deadline* must be a unix timestamp of the future.

After the successful execution of the *addLiquidity* function, you can go to your Metamask wallet and check if you have less tokens which means that the tokens have been transferred into the liquidity pool.

3.3 CLP with sufficient amount of token A

Besides a DEX with a liquidity pool and its corresponding tokens, a CLP is required for our FLA. We deliberately constructed an exploitable CLP (inspired by bZx) to conduct our FLA with. Within our FLA the task of the CLP is to lend tokens against another token as collateral. Thus, the CLP must have enough tokens to lend.

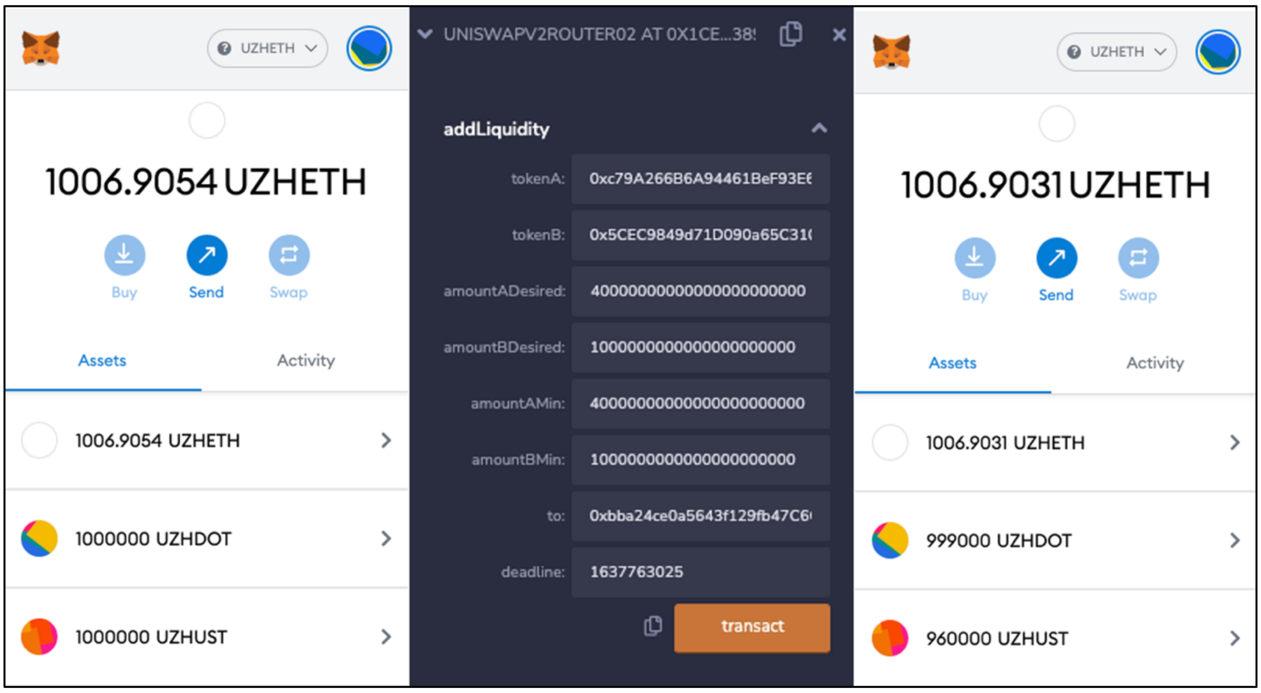
Concerning the deployment you will find the code in *flash-loan-attack/collateral-loan*.

Go to line 10 of *CollateralLoan.sol* and make sure that this address is set to the DEX address on which you have created the liquidity pool for your two tokens. Also change the address of your second DEX on line 11 if you have deployed one.

Deploy this smart contract and transfer enough token A to its address. Enough token A means more than you want to lend later against a collateral which is token B. In our example case we sent 500'000 UZHDOT to our address of *CollateralLoan.sol.*

3.4 FLP with sufficient amount of token A

Furthermore, for our FLA a Flash Loan Provider (FLP) must be established. The FLP will lend tokens to the borrower as a flash loan.



UZHETH

1006.9054 UZHETH

Buy Send Swap

Assets Activity

1006.9054 UZHETH

1000000 UZHDOT

1000000 UZHUST

UNISWAPV2ROUTER02 AT 0X1CE...38!

addLiquidity

tokenA: 0xc79A266B6A94461BeF93E6

tokenB: 0x5CEC9849d71D090a65C31(

amountADesired: 40000000000000000000000

amountBDesired: 1000000000000000000000

amountAMin: 40000000000000000000000

amountBMin: 1000000000000000000000

to: 0xbba24ce0a5643f129fb47C6

deadline: 1637763025

transact

UZHETH

1006.9031 UZHETH

Buy Send Swap

Assets Activity

1006.9031 UZHETH

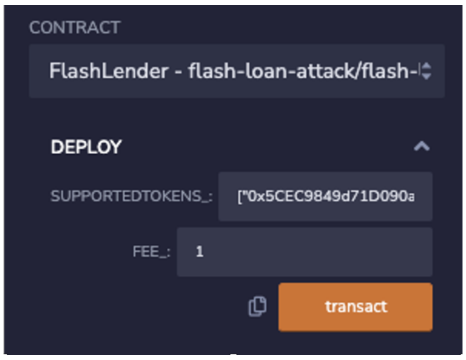
999000 UZHDOT

960000 UZHUST

Figure 4 shows example parameters for the *addLiquidity* function and its successful execution from left to right

For the deployment, you can find the corresponding smart contract at *flash-loan-attack/flash-loan/FlashLender.sol.*

Set the array of supported tokens which at least must include the address of our token to lend (token A), insert the fee which is measured in 1/10000 units and deploy it. Figure 5 shows that we set UZHDOT as the only supported token and a fee of 0.0001% for taking out a flash loan.


CONTRACT

FlashLender - flash-loan-attack/flash-l

DEPLOY

SUPPORTEDTOKENS_: ["0x5CEC9849d71D090a

FEE_: 1

transact

Figure 5 displays example parameter for the deployment of *FlashLender.sol*

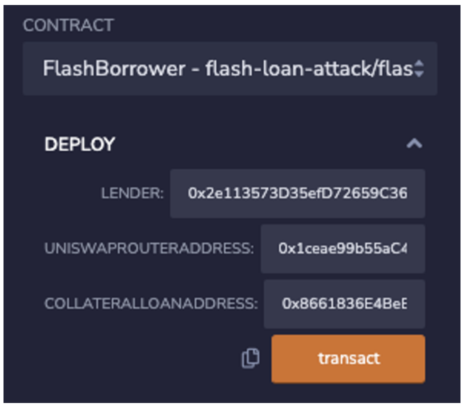
Next, send enough of token A from your Metamask wallet to the address of *FlashLender.sol.* In our case we sent 99'000 of UZHDOT to our address of *FlashLender.sol*

3.5 Attacking Contract

The origin of our FLA is the attacking contract which acts as the flash loan borrower and must be created before the attack can happen. At the time the attacking contract has received the flash loan it executes the attack.

Regarding the deployment you can find the corresponding smart contract called *FlashBorrower.sol* at *flash-loan-attack/flash-loan.*

Deploy this protocol with the address of your *FlashLender.sol* as *LENDER,* the address of the DEX on which you pre-set your liquidity pool as *UNISWAPROUTERADDRESS* and the address of your *CollateralLoan.sol* as *COLLATERALLOANADDRESS.* Figure 6 displays what parameter we used for our *FlashBorrower.sol* deployment.


CONTRACT

FlashBorrower - flash-loan-attack/flas

DEPLOY

LENDER: 0x2e113573D35efD72659C36

UNISWAPROUTERADDRESS: 0x1ceae99b55aC4

COLLATERALLOANADDRESS: 0x8661836E4BeE

transact

Figure 6 illustrates example parameter for the deployment of *FlashBorrower.sol*

4 EXECUTION OF THE FLA

This chapter acts as a hands-on guide to help you reproduce a FLA on the UZHETH network by yourself. Note that you first must consider the preconditions of chapter 3 otherwise the FLA won't be successful.

Before starting the execution of the FLA make sure that the *setIsFlashLoanAttackPossible* attribute on the *CollateralLoan.sol* protocol is *true*. If not change it via executing the *setIsFlashLoanAttackPossible* function with the parameter *true.*

At this point we are ready to conduct a FLA. Figure 7 depicts an overview of the FLA process and its consecutive execution steps

The entry point of our FLA is implemented within the *FlashBorrorw.sol* acting as the attacking protocol. Executing the first step of our FLA, we therefore invoke the *flashLoanAttack* function on the *FlashBorrower.sol*. Use the address of your token A for *tokenToFlashLoan*, an integer as *amountToFlashLoan* and your address of token B for *tokenToSwap* as function parameters. The middle part of Figure 8 shows that our group took the address of UZHDOT as *tokenToFlashLoan*, the address of UZHUST as *tokenToSwap* and an *amountToFlashFloan* of 10'000 UZHDOT.

The execution of the *flashLoanAttack* function will then by construction automatically trigger the other steps of our already deployed precondition smart contracts. After the execution went through you should have a positive delta of token A in your Metamask wallet and therefore successfully conducted our FLA as shown in Figure 8 with the increased amount of UZHDOT in our wallet.

In step 2 we request the flash loan for *tokenToFlashLoan* which is UZHDOT in our case.

After having received the flash loan step 3 and step 4 will be triggered where 90% of the loaned amount will be swapped to *tokenToSwap* which is UZHUST in our case. At this point we have both *tokenToFlashLoan* and *tokenToSwap* in our wallet.

Step 5 then takes a loan of *tokenToFlashLoan* out of the CLP against the collateral of *tokenToSwap.* Our CLP will solely use the liquidity pool from one DEX, i.e. Uniswap, for the valuation of *tokenToFlashLoan* and *tokenToSwap*. Since we have manipulated this liquidity pool beforehand in step 3 and step 4 by swapping many tokens, we obtain much more of *tokenToFlashLoan* than it is worth.

Step 6 is the last step where we pay back the flash loan and transfer the positive delta to our Metamask wallet. It is our intention that we never pay back the collateral loan which is probably being liquidated at some from the CLP. Otherwise, our FLA won't work.

In Table 3 we provide the code lines where the steps are being executed.

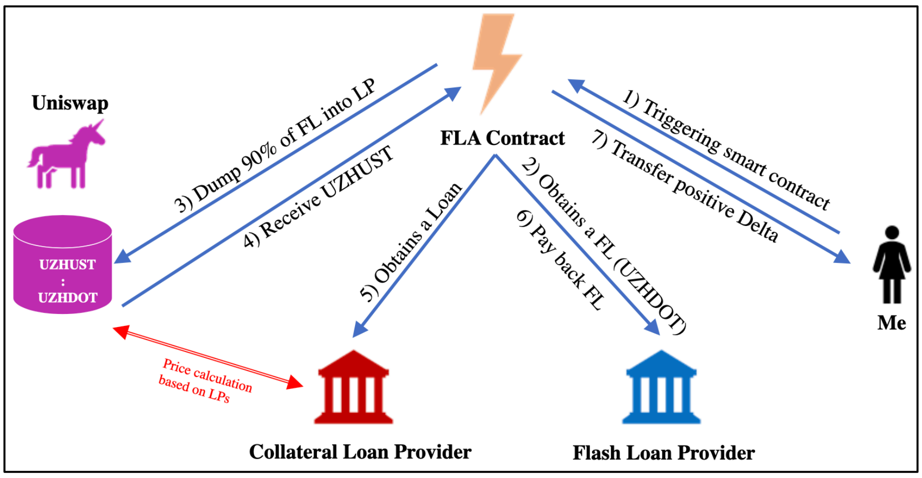
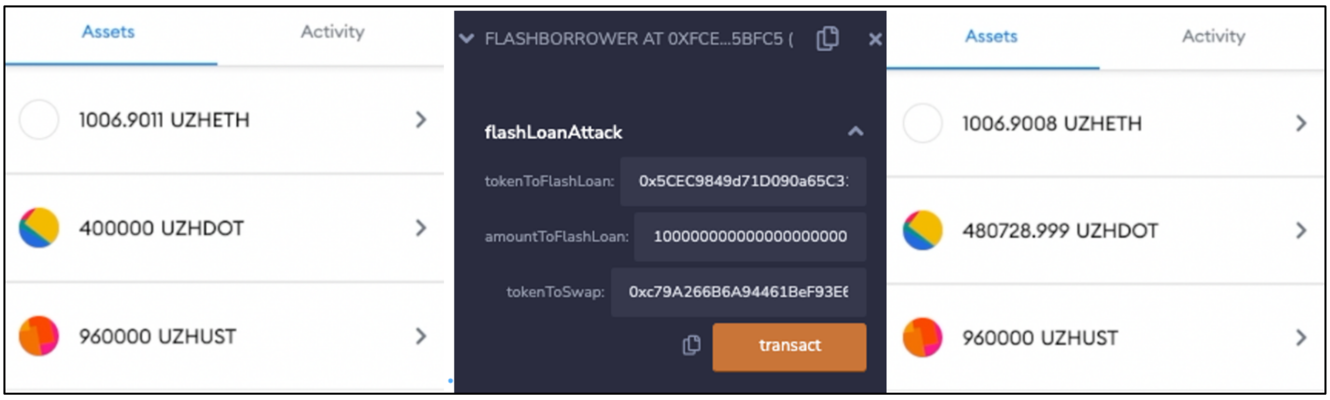


Figure 7 displays the procedure of our FLA

Steps	Smart Contract	Line numbers and its context
Step 2	FlashBorrower.sol	61 (request flash loan)
Step 2	FlashLender.sol	20-23 (pay out the flash loan)
Step 3 & Step 4	FlashBorrower.sol	27-40 (trigger a swap on Uniswap)
Step 5	FlashBorrower.sol	42-47 (take out the collateral loan from the CLP)
Step 6	FlashLender.sol	24 (pay back the flash loan)
Step 6	FlashBorrower.sol	64 (transfer the positive delta to our Metamask wallet)

Table 3 shows where each individual step is executed in our code


Assets Activity

1006.9011 UZHETH

400000 UZHDOT

960000 UZHUST

FLASHBORROWER AT 0XFCE...5BFC5 (

flashLoanAttack

tokenToFlashLoan: 0x5CEC9849d71D090a65C3:

amountToFlashLoan: 100000000000000000000

tokenToSwap: 0xc79A266B6A94461BeF93E€

transact

Assets Activity

1006.9008 UZHETH

480728.999 UZHDOT

960000 UZHUST

Figure 8 illustrates the successfully conducted FLA and our positive delta of 80’728.999 UZHDOT

5 PREVENTION OF THE FLA

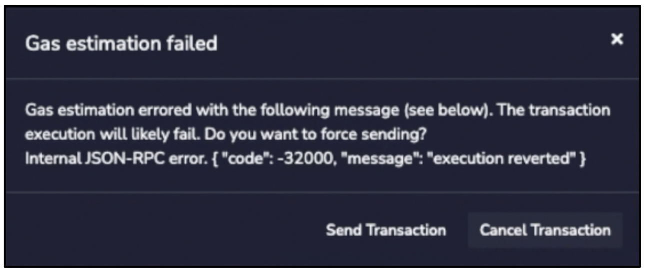
Since our presented FLA is based on the oracle manipulation principle our prevention methods focus on how to make the CLP's oracle less vulnerable to manipulations. Basically, our approach is to use averaged token prices from multiple data providers. In this section we document two such methods.

5.1 Averaging Price Function

Our first prevention method consists of a simple function which averages the token prices from multiple DEXs. If the vulnerable CLP uses this function to calculate the number of tokens to lend against the collateral, the attacker must know and have manipulated every single DEX simultaneously for still being able to execute a successful attack. The more independent DEXs you use in the averaging price function the less vulnerable a CLP is against attacks.

For simplicity reasons, we agree on only using two different DEXs for this function, namely Uniswap and Sushiswap. The averaging function is placed between line 26 and 32 in *CollateralLoan.sol*.

Before testing the prevention make sure that the *setIsFlashLoanAttackPossible* value on *CollateralLoan.sol* is *false*. If not modify it to *false* for a successful activation of this prevention method. Next, run step 1 of the FLA as described in chapter 4 and you should get an error message as in Figure 9. Then, the transaction has been reverted because you have not had the means to pay back the flash loan meaning the prevention was successful.


Gas estimation failed

```
Gas estimation errored with the following message (see below). The transaction execution will likely fail. Do you want to force sending?
Internal JSON-RPC error. { "code": -32000, "message": "execution reverted" }
```

Figure 9 demonstrates the successful FLA prevention via the averaging price function

5.2 Chainlink Price Feed

Chainlink 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. Chainlink's pre-built decentralized price feeds provide DeFi applications a real-time stream of financial market data, including exchange rates for cryptocurrencies, stablecoins, commodities, indices, stocks, fiat currencies, and other key financial datasets. Data feeds are available on networks such as EVM-compatible networks, Solana, and Terra. The solution is that price information of an asset needs to come from decentralized oracles, to be more specific from decentralized Chainlink Price Feeds.

In contrast to the Ethereum blockchain, which operate as a single monolithic network with a one consensus mechanism (currently proof-of-work), the Chainlink Network is a decentralized network consisting of numerous independent oracle networks running the same software but operating fully independent as shown in Figure 10. This leads to the possibility of horizontal scaling as any number of oracle networks can operate in parallel.

A common reaction of DeFi protocols, which have hit by FLAs, is to integrate such price feeds from a Chainlink network such that their price calculations is not anymore depending on liquidity pools.

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. However, we provided code for both options in *flash-loan-attack/chainlink.*

Opening *local-deployment.sol* it is split into two parts. The first part is used to set up a local Chainlink node using Docker and the second part is the smart contract which requests prices via our running node.

While executing the first part in your command line, the node's directory gets created, its environment configured and the Docker parity image run. Note that you must have installed Docker beforehand on your computer for running this script.

Next, we must deploy *APIConsumer.* In there, the *jobId* variable represents the dxFeed Price Oracle but you can use any Chainlink oracle that has a job that can return a bytes32. Our Chainlink node uses this job for building a Chainlink request and fetch price data from an API. The *oracle* variable represents the address of our locally deployed Chainlink node which will receive fetched data from the API via the Chainlink request. In our example we did an Ether price request.

For a successful prevention the last step would be to invoke the price request function of *APIConsumer* within the CLP's lending amount calculation method. Since we did our FLA with two newly created tokens on UZHETH we won't receive data for these tokens from our real API. Therefore, this prevention method cannot be fully tested on UZHETH.

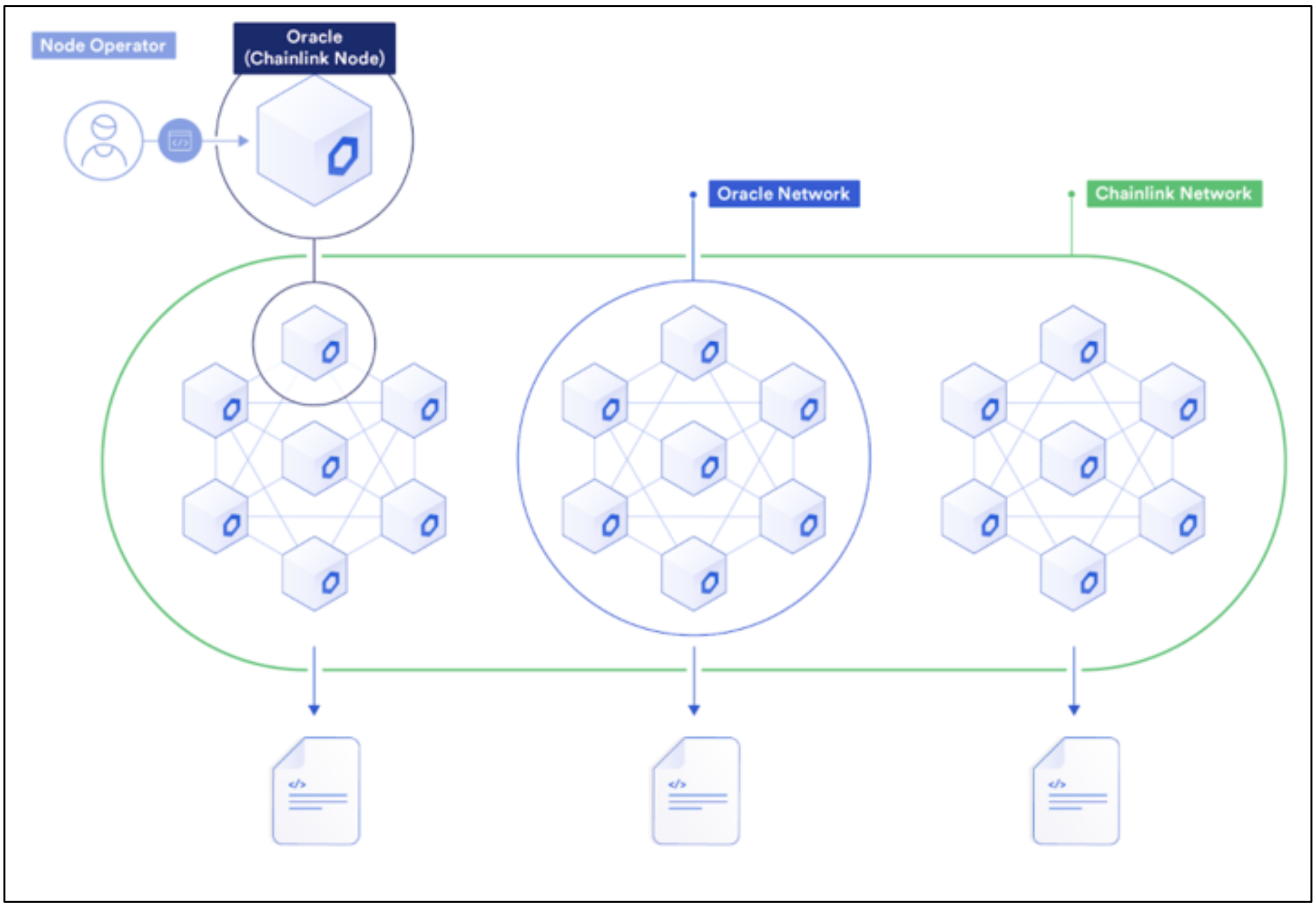


Figure 10 visualizes the conceptual structure of a the Chainlink Network

6 METHODOLOGY AND ERROR HANDLING

This chapter documents how we approached the creation and deployment of the FLA itself and its preconditions. It also describes occurred errors during the process and our solutions.

6.1 ERC20 Tokens

Before we deployed our two tokens, we copied and adapted the provided UZHETH template with a customized name for each token. Thus, we had no deployment errors.

6.2 Decentralized Exchange (DEX)

As our group has decided on using Uniswap we cloned its source code from GitHub and first tried to deploy the Uniswap Factory and Uniswap Router contracts and its dependencies. The Router contract is dependent on a WETH token which we created manually according to the UZHETH token template. After all deployments were finished, we tried to add a liquidity pool where we had three major errors.

Having invoked the *addLiquidity* function on *UniswapV2Router02.sol* we received the error message *Internal JSON-RPC error. {"code": -32000, "message": "execution reverted"}*. We solved it by following an online article which mentions that we first should replace the *INIT_CODE_PAIR_HASH* attribute on line 24 of the file *UniswapV2Library.sol* with the *INIT_CODE_PAIR_HASH* of our deployed *UniswapV2Factory.sol*.

The next error message was *Internal JSON-RPC error. {"code": 3, "message": "execution reverted: TransferHelper:transferFrom: transferFrom failed"}* which related to the missing approval for the Uniswap Router to take our tokens out of our Metamask wallet to add it to its liquidity pool. So, we first had to approve the Router's address to take the number of tokens we wanted to add from our Metamask wallet. For further information see chapter 3.2.

The third problem was *Internal JSON-RPC error. {"code": 3, "message": "execution reverted: ds-math-sub-underflow"}* which we mapped to the 2 decimal digits of the UZHETH token template we used for deployment for our tokens. The solution was to redeploy the two tokens with 18 decimals each.

After having solved these problems and conducted a redeployment of the adapted smart contracts and its dependencies we were finally able to successfully setup a liquidity pool.

6.3 Collateralized Loan Provider (CLP)

With the help of an online article, we managed to create our vulnerable CLP. We took *loan.sol* as template for our newly created *CollateralLoan.sol* contract.

Our only problem was how to fetch the current token prices from Uniswap for calculating the amount to lend in relation to the collateral value. Our solution was to manually implement a price determination function according to the liquidity pool reserves.

6.4 Flash Loan Provider

First our approach was to deploy AAVE because their code is open source and already provide a flash loan functionality. We found a relatively simple documentation on how to deploy an AAVE application in remix IDE. Unfortunately, these documentations always use Ropsten or another Ethereum Testnet which does not really help. During the deployment process on UZHETH we faced a lot of problem which we could not really trace back. Therefore, we decided on implementing the flash loan provider functionality from scratch with the help of an online article. Basically, we copied *FlashLender.sol* as template and adapted the code for creating our version of *FlashLender.sol*.

6.5 Attacking Contract

With the help of the same article as for the Flash Loan Provider we managed to create an attacking contract called *FlashBorrower.sol*. We refactored the code to solidiy 0.8.0 and adapted its content to the steps of our flash loan attack. Plus, at the end of the *flashLoanAttack()* we added *'IERC20(token). transfer (msg.sender, IERC20(token).balanceOf(address(this)));'* to finally send the exploited tokens to the attacker.

During the testing process we received the error *Internal JSON-RPC error. {"code": -32000, "message": "execution reverted"}* inside of *flashBorrow()* because of the line *IERC20(token).approve(address(lender), _allowance + _repayment);*. At this point the *FlashBorrower.sol* does not own *_allowance* + *_repayment* tokens and therefore cannot approve this amount to the *FlashLender.sol*. We had to move this part to *onFlashLoan()* on because only at this point the *FlashBorrower.sol* has already received the flash loan and can approve it.

7 SUMMARY

The group successfully executed an oracle manipulation attack on the UZHETH network and pointed out possible prevention methods. Our FLA exploits a CLP which implemented a price calculation based on liquidity pools of Uniswap V2. Additionally, the group introduced some of the most robust prevention techniques such as average price calculation and the introduction of Chainlink price feeds. Since several FLAs were conducted in the last 12-24 months the topic of how to prevent these type of attacks remains highly relevant. Oracle manipulations are not the only type of FLAs, yet it is currently the most popular type of attack. The approach of integrating Chainlink has shown how a decentralized data source can protect a DeFi protocol from exploits.

8 AUTHOR CONTRIBUTIONS

All authors conceived and designed the project idea. P.E. and N.C. performed the FLA creation whereas M.K focused on the Chainlink node solution including local deployment and using a faucet testnet. The divided work each contained the literature research, programming tasks, responsibility for the chapters in this report and recording of the corresponding video part. All authors discussed and reached the conclusions. All authors revised and accepted the final version of this document.

9 REFERENCES

Glassnode Insights - On-Chain Market Intelligence. (n.d.). *Glassnode Insights - Defi*. Glassnode Insights - On-Chain Market Intelligence. Retrieved November 29, 2021, from https://insights.glassnode.com/tag/defi/.

Using data feeds (EVM): Chainlink documentation. Chainlink Developers. (n.d.). Retrieved November 29, 2021, from https://docs.chain.link/docs/get-the-latest-price/.

PreventFlashLoanAttacks. (n.d.). Home. Retrieved November 29, 2021, from https://preventflashloanattacks.com/.

GitHub. 2021. GitHub - Uniswap/v2-core: Core smart contracts of Uniswap V2. [online] Available at: <https://github.com/Uniswap/v2-core> [Accessed 1 December 2021].

Docs.uniswap.org. 2021. Factory | Uniswap. [online] Available at: <https://docs.uniswap.org/protocol/V2/reference/smart-contracts/factory> [Accessed 1 December 2021].

GitHub. 2021. GitHub - Uniswap/v2-periphery: 🦄 Peripheral smart contracts for interacting with Uniswap V2. [online] Available at: <https://github.com/Uniswap/v2-periphery> [Accessed 1 December 2021].

Docs.uniswap.org. 2021. Router02 | Uniswap. [online] Available at: <https://docs.uniswap.org/protocol/V2/reference/smart-contracts/router-02> [Accessed 1 December 2021].

Medium. 2021. The bZx Attacks—What Went Wrong and the Role Oracles Played in the Exploits. [online] Available at: <https://medium.com/meter-io/the-bzx-attacks-what-went-wrong-and-the-role-oracles-played-in-the-exploits-264619b9597d> [Accessed 1 December 2021].

Youtube.com. 2021. [online] Available at: <https://www.youtube.com/watch?v=U3fTTqHy7F4> [Accessed 1 December 2021].

GitHub. 2021. eattheblocks/screencast/229-fork-uniswap at master · jklepatch/eattheblocks. [online] Available at: <https://github.com/jklepatch/eattheblocks/tree/master/screencast/229-fork-uniswap> [Accessed 1 December 2021].

Popadić, A., Popadić, A., Workshop, M. and Workshop, M., 2021. Uniswap v3 Explained - Everything You Need to Know. [online] MVP Workshop. Available at:

<https://mvpworkshop.co/blog/uniswap-v3-explained-all-you-need-to-know/> [Accessed 1 December 2021].

Blockchain.news. 2021. How to Build a Decentralized Exchange (DEX) Like Uniswap in Less than One Hour. [online] Available at: <https://blockchain.news/wiki/how-to-build-an-uniswap-exchange> [Accessed 1 December 2021].

Vomtom.at. 2021. Uniswap v2 (as a Developer). [online] Available at: <https://vomtom.at/how-to-use-uniswap-v2-as-a-developer/> [Accessed 1 December 2021].

Stackexchange.com, H. and Ortiz, C., 2021. How can you get the price of token on Uniswap using solidity?. [online] Ethereum Stack Exchange. Available at: <https://ethereum.stackexchange.com/questions/91441/how-can-you-get-the-price-of-token-on-uniswap-using-solidity/94173> [Accessed 1 December 2021].

EatTheBlocks. 2021. How To Perform Custom Ethereum Flash Loans Using Solidity (ERC 3156 Standard) - EatTheBlocks. [online] Available at: <https://eattheblocks.com/how-to-perform-custom-ethereum-flash-loans-using-solidity-erc-3156-standard/> [Accessed 1 December 2021].

Market.link. 2021. Chainlink Market. [online] Available at: <https://market.link/nodes/eb5c92a8-6093-4657-9a68-a6d10719946e/integrations?network=1> [Accessed 1 December 2021].