

**Blockchain Programming**

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

Maximilian Kiefer (XX-XXX-XXX)

Pascal Emmenegger (XX-XXX-XXX)

Nicola Crimi (12-748-612)

Affiliation

Date: 01.12.2021

**ABSTRACT**

The goal was to successfully conduct a flash loan Attack within the UZHETH Network. Out of several types of flash loan attacks, the team decided to conduct an oracle manipulation. The intention was to manipulate the price calculation function of a DeFi protocol that could execute collateralized loans. Finally, the question of how to prevent these kinds of attacks should be answered within the scope of this project.

## Keywords

flash loan attack, oracle manipulation, liquidity pools, prevention

# Introduction

Flash loan attacks (FLA’s) are a very important issue as many blockchain networks are subject to these kinds of attacks and are losing millions of US dollars. A FLA occurs when the borrower uses the markets as the loan is taking place, driving the value of a token underwater (there are several ways of doing so, which is explained later in this paper), 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 Protocols' 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 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.

# Set UP for A Flash Loan attack

## Programming and smart contracts

We are using UZHETH network smart contracts, and the programming being done is with the Solidity language. Furthermore, the Metamask extension is used as a wallet. All the programming has been done within the remix IDE.

## Deploying a decentralized exchange (DEX)

In order to conduct a FLA within the UZHETH network it is necessary to have at least one working DEX. The team decided on deploying the Uniswap V2 (without GUI). In consecutive steps, this DEX is used to manipulate its liquidity pools. The Uniswap V2 code is openly accessible on GitHub.

Smart Contract Name/address: xyz.sol/XXXXXXX

## Deploying DeFi Protocols

Besides a DEX, the team needed a (at least one) DeFi Protocol which was able to issue un-/collateralized (flash) loans. For the sake of simplicity, it decided on deploying two separate protocols. One protocol which was able to issue flash loans (FlashLender.sol) and one protocol which issues loans only against collaterals (CollateralLoan.sol). This protocol bases its price calculation on the DEX (see 2.2).

Smart contract name/address: xyz.sol/XXXXXXX

Smart contract name/address: xyz.sol/XXXXXXX

## Implementing and deploying a flash loan attack

The attack itself is implemented within the FlashBorrorw.sol smart contract which is responsible for the following steps:

1. Obtaining a flash loan from the FlashLender.sol contract
2. Swap a portion of the flash loan on the DEX and shift the liquidity ratio to the favour of the attacker (FlashBorrower.sol)
3. The tokens obtained by the swap (in our case USTs) swapping against a collateral (DOTs)
4. Pay back the flash loan which was initiated in step 1
5. Transfer the (positive) delta into the Metamask

Smart contract name/address: xyz.sol/XXXXXXX

## Minting ERC20 Tokens

For the oracle manipulation two tokens are needed. The group minted two ERC20 tokens (UZHDOT.sol, UZHUST.sol) with a total supply of 1’000’000.

Smart contract name/address: xyz.sol/XXXXXXX

Smart contract name/address: xyz.sol/XXXXXXX

# Conceptual overview

This section is dedicated to a general overview of the domain of flash loan attacks, about the recent history of attacks and about possibilities to prevent these kinds of attacks.

## Definition of a Flash Loan

A flash loan is a relatively new possibility of uncollateralized lending offered by a DeFi protocol. Furthermore, a flash loan is only valid within one blockchain transaction. 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. (Source: <https://preventflashloanattacks.com/>)

## Types of flash loan attacks

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

Since most of the conducted flash loan attacks in 2020 were oracle attacks, this project 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 when it comes to these types of manipulations:

1. Taking out a massive loan (e.g. token A) from a protocol supporting flash loans. In our case from the CollateralLoan.sol contract
2. Swapping token A for token B on a DEX (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.

## Historic overview of FLAs

Since DeFi protocols allow flash loans (ca. end of 2019/early 2020) many protocols suffered from all kinds of flash loan attacks. In the following a brief overview:

Table 1 Overview of biggest FLAs in 2020

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

The project focuses on the bZx (2) attack which was one of the first oracle attacks and which is later fixed with the help of a Chainlink integration.

# Hands-on 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.

## Uniswap

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.

# Prevention methods

## 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.

## Other methods and examples

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.

This code shows a prevention example:

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;

# Figures and tables

## General appearance

Make sure that all figures, tables, graphs and line drawings are clear and sharp and of the highest quality. Lines should be thick enough to allow proper reproduction. **Also in figures: use embedded arial font type only.**

Diagrams, graphics and photographs should be in **gray scale or in colour** of excellent quality with good contrast.

When preparing figures and tables, make sure that all lettering inside the figure is no smaller than the specified size of the paper text, i.e., **10 point**. Do not include any headlines in the diagrams, graphics or tables. All headlines should be written separately. See the examples below. Do not use different colours in diagrams. If you use a bar graph, please use a pattern that will appear clearly in black and white. Use different patterns instead of colours, as the colours will not provide sufficient contrast when printed in black and white.

If necessary add a source below the diagram. Do not add any kind of background color in the graph. The background should always be white.

## Numbering, captions and positioning

Number the figures separately from the maps and tables e.g., Figure 1, Figure 2, Figure 3; Table 1, Table 2, Table 3. Map 1, Map 2, Map 3 etc. Use (a), (b), (c) to distinguish individual subjects in a composite figure. See Figures 1 and 2 for examples of figure and caption placement

the paper. Begin the caption with a capital letter and end with a full stop. Always refer to figures as ‘Figure’ and not Fig. Place the figure or table on the text page as close to the relevant citation as possible, preferably at the top of a column. If a figure or table is too large to fit into one column, it may be centred across both columns at the top or the bottom of the page.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Country | 65–79 | | Over 80 | |
| Females | Males | Females | Males |
| Sweden | 45 | 34 | 15 | 9 |
| United Kingdom | 34 | 25 | 10 | 5 |
| France | 45 | 38 | 19 | 11 |
| Germany | 28 | 32 | 21 | 17 |
| Spain | 31 | 24 | 19 | 12 |

**Table 1** is an example of how a table or figure may be placed in a column, preferably in the beginning of a column.

# summary of Errors

1. Token issue: 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" }¨

2. Flash loan: 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

3.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

4.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

5.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).

6.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.

7.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.

8.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. We have issues in either

**8 AUTHOR CONTRIBUTIONS**

The report must have an authorship statement at the end. An example is the following

AUTHOR CONTRIBUTIONS

Maximilian Kiefer worked on the chain-link node solution from local deployment and using a faucet testnet which is demonstrated in the video and documentation.

# 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].

solidity?, 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].