Exploiting Uniswap: from reentrancy to actual profit

OPENZEPPELIN SECURITY | JULY 28, 2019

Security Audits

Don’t you just love socks?. At OpenZeppelin, we all have a pair of super-cool, colorful, nerdy socks which we’ve been proudly showing off, but we knew we had to sport something new for the upcoming season! On the lookout for the perfect pair, we serendipitously ran into this tweet about Unisocks, and it was love at first sight 😍.

After a few quick searches, we found out that the price of the Unisocks was governed by a decentralized exchange called Uniswap, a set of smart contracts coded in Vyper. Inevitably, the naive side of our security-researcher minds spoke up: “Well, if a smart contract is backing this thing, **we might as well try getting us some free socks**!”. Aaand that was the beginning of a week-long journey hunting down a free pair of socks…

… which we never got.

Though, we did learn something cool: **how to exploit and profit from any Uniswap exchange that uses an ERC777 token**. And we’re sharing that with you now.

Here is a short disclaimer before getting our hands dirty. What you’re about to see is a proof-of-concept exploit of an already public, disclosed, and acknowledged vulnerability in Uniswap related to reentrancy attacks; we wouldn’t be publishing this otherwise. The first-ever attack vector on

OpenZeppelin

Quick intro to Uniswap

Uniswap is a public, open-source protocol to exchange tokens in Ethereum. In Uniswap, there is a separate exchange contract for each token. While it was designed to seamlessly work with ERC20 tokens, token listing is open and free. So there's nothing preventing you from registering a Uniswap exchange for a ERC20-compliant token with extended functionality.

Liquidity and exchange rate

Someone has to provide tokens and Ether (a.k.a., liquidity) to the exchange once a token exchange is created. Liquidity plays a fundamental role in Uniswap, since the price at which the assets are exchanged in each trading operation depends on the relative size between the Ether and token reserves, as well as the amount with which an incoming trade shifts their ratio.

For example, selling tokens to the exchange increases the size of the token reserve, while decreasing the size of the Ether reserve. As the reserve ratio is changed, in the next token-sale operation the exchange will pay less ETH for the same amount of tokens. You can read more about the pricing mechanism in the Uniswap docs.

Exchanging tokens for ETH in Uniswap

Alright, let's get down to business. Consider a live Uniswap exchange, all healthy and thriving on mainnet. Say that this exchange already has reserves both in tokens and Ether, so anyone can normally trade in it.

Now, Alice has some spare tokens that she wants to sell for Ether, and decides to exchange them in Uniswap for a convenient token-to-Ether price ratio.

She calls the tokenToEthSwapInput function of the exchange, having previously approved the tokens, stating the amount of tokens she's willing to sell. This function will in turn call the private tokenToEthInput function, mainly in charge of:

1. Calculating the exact exchange rate (calling the private getInputPrice function)
2. Sending Alice the corresponding Ether

Uniswap + ERC777

Things get far more interesting if we tweak the previous scenario a little bit. Now, let's assume that:

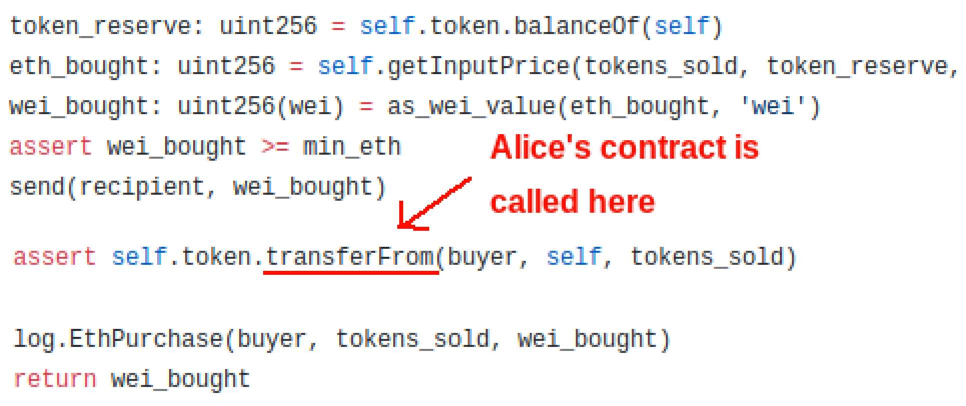
1. The token being traded in Uniswap is not a simple ERC20 but an ERC20-compliant ERC777 (doesn't have to comply fully with the 777 spec, but for simplicity, let's say it is).
2. Alice has gone to the dark side 👿 and will execute the transfer not from an externally-owned account but through a malicious contract.

If this is the first time you've heard of ERC777, don't worry! There's just **one fundamental feature of the ERC777 that you need to be aware of: the ERC777 hooks**. In any transfer of tokens, an ERC777 contract is basically going to:

1. Call the sender of tokens – in our case, Alice.
2. Execute the transfer (i.e., swap balances and reduce allowances if appropriate)
3. Call the recipient of tokens

For a real example, checkout the transferFrom function in OpenZeppelin's implementation of the ERC777. It's worth highlighting that in the case of the transferFrom function, the recipient is not called if it's not registered in the ERC1820 registry.

So thanks to the **ERC777 hook that's executed *before* the actual transfer of tokens**, the sender (Alice's contract) gets called and can therefore execute code. Getting back to Uniswap's tokenToEthInput function, it will now look like this:



```
token_reserve: uint256 = self.token.balanceOf(self)
eth_bought: uint256 = self.getInputPrice(tokens_sold, token_reserve,
wei_bought: uint256(wei) = as_wei_value(eth_bought, 'wei')
assert wei_bought >= min_eth
send(recipient, wei_bought)

assert self.token.transferFrom(buyer, self, tokens_sold)

log.EthPurchase(buyer, tokens_sold, wei_bought)
return wei_bought
```

This means that now the tokenToEthInput function of the exchange actually:

1. Calculates the exact exchange rate (calling getInputPrice)
2. Sends Alice's contract the corresponding Ether, reducing the exchange's ETH reserves
3. **Calls Alice's contract**
4. Transfers Alice's tokens to the exchange, increasing the exchange's token reserves

In (3), Alice gets total control of the situation. It's fundamental that you understand that at this point:

- The exchange's **ETH reserves** were already **decreased**
- The exchange's **token reserves** were **not yet increased**
- Alice's contract gets to decide what to do now 😈

Reentrant microtrading in Uniswap

Clever enough, Alice can leverage the call received to **reenter the Uniswap exchange** by calling the tokenToEthSwapInput function again 😱.

In this second token-buy call, **the ETH reserves will be lower, but the token reserves will be the same**. This means that this second batch of tokens is going to be **exchanged for just a little more ETH than what they should be**. Why is that? Because math.

This is the formula governing the price at which the exchange will buy the tokens Alice is selling:

$$\begin{aligned}&where\\ &\text{Ts}:\ amount\ of\ tokens\ being\ sold\ by\ caller\\ &\text{Tr}:\ current\ reserve\ of\ tokens\\ &\text{ETHr}:\ current\ reserve\ of\ Ether\end{aligned}$$

Under normal operation, after subsequent regular token sales, the reserve of tokens would go up (denominator grows larger), and the reserve of Ether would go down (numerator shrinks). Therefore, the amount paid for the tokens must decrease after each round of sales, which makes sense.

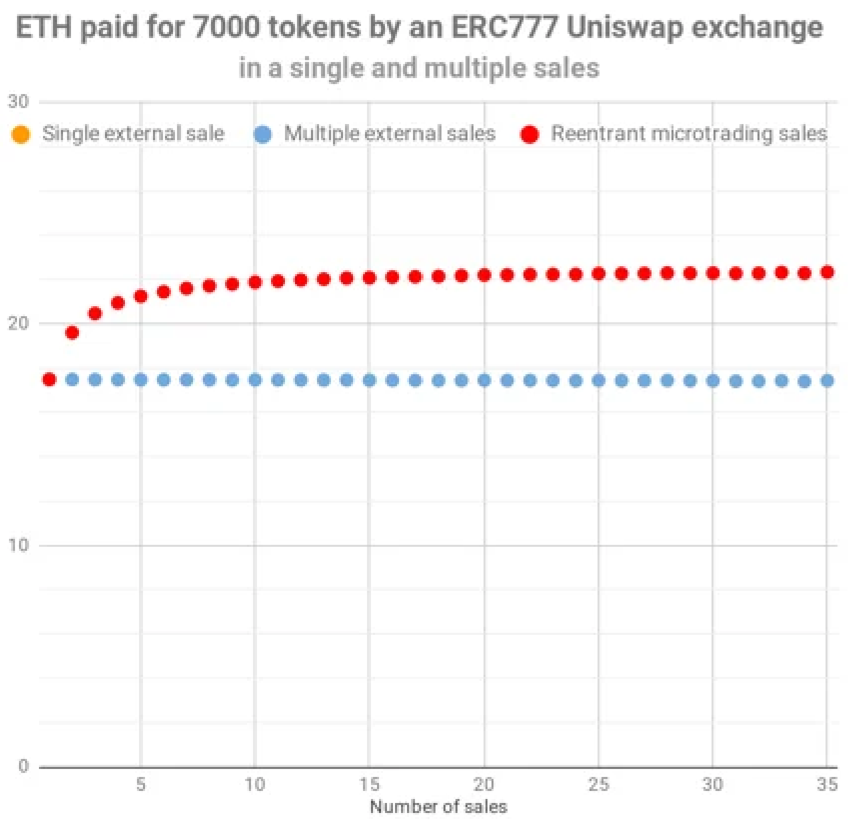
In contrast, by exploiting the reentrancy, Alice is going to effectively prevent the amount of tokens in reserve from increasing, turning the denominator of the equation into a constant. Note that the amount of ETH in reserve is still going to get lower (i.e., the numerator will be smaller in each reentrant call). Still, in the long run (after several reentrant iterations), Alice is going to be able to make a substantial profit. In fact, the more iterations, the better. That's why we dubbed this the **"reentrant microtrading" attack**.

Reentrant microtrading in action

Let's think of three main token-sale scenarios which should help make it all clearer. In all of them, the exchange starts with 30 ETH and 5000 tokens in reserves. As you're about to see in the plot and exploit's code, the attacker is able to **increase profits by more than 27%**.

Scenario	Description	ETH received
A) Legitimate trading	A normal user executes a single transaction to sell 7000 tokens.	~17 ETH
B) Legitimate trading	A normal user executes 35 transactions, each time selling 200 tokens. 7000 tokens are sold in total.	~17 ETH
C) Exploiting	An attacker (through a contract) executes a single transaction leveraging the reentrant microtrading strategy. 35 iterations are done, each time selling 200 tokens. 7000 tokens are sold in total.	~22 ETH

How does it all look like? Like this:



Source

Each dot represents the amount of Ether paid by the Uniswap exchange given a fixed number of sales. For instance, in 20 reentrant sales of 350 tokens each, the reentrant microtrading strategy (in red) makes ~22.192 ETH, whereas if the 7000 tokens were sold by externally calling the tokenToEthSwapInput function 20 times (selling 350 tokens in each call), the profit would only be ~17.44 ETH (in blue).

Note that the three strategies start at the same point (~17.47 ETH for 7000 tokens), but as the number of calls grows larger, **the reentrant strategy shows a substantial difference in profit**. For 35 calls, while the legitimate trading (in blue) makes ~17.418 ETH, the microtrading strategy (in red) results in ~22.324 ETH. Also bear in mind that the dotted blue curve, representing multiple external sales, is not constant, but it has a minor slope (unnoticeable due to the y-axis scale) indicating that it is less and less profitable to make multiple external sales of tokens. To confirm this, checkout the numbers in the results.

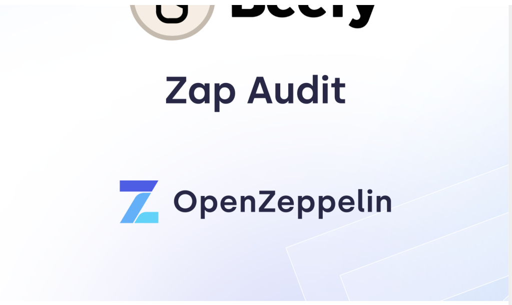
What's most interesting here (and fun) is **understanding how to properly attack that vulnerability to make real profit in Uniswap**.

Hoping that the above analysis shed some light into that, we leave you with the long-promised working proof-of-concept exploit, covering all scenarios described before.

Running it should be pretty much straightforward if you follow the instructions in the README file. To dive deeper into the code, make sure you check out the uniswap.exploit.js file and the Attacker.sol contract. To see the results in CSV format, go to the results folder.

That's all hackers! OpenZeppelin's security research team keeps digging, so expect more news from us soon!

Related Posts



Beefy Zap Audit

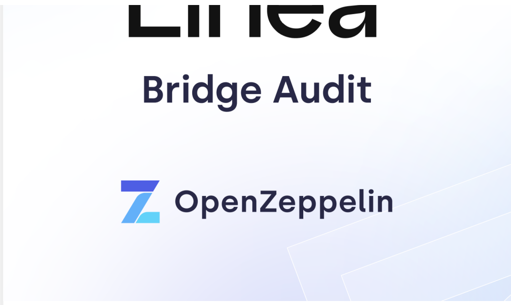
BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits



Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits