This is a horror story.

The Challenge

Like any normal person, I spend a lot of time lurking in the #support channel of the Uniswap Discord.
(Disclosure: Uniswap is a portfolio company of Paradigm.)

On Wednesday afternoon, someone asked whether it was possible to recover Uniswap liquidity tokens that had been accidentally sent to the pair contract itself.

My initial thought was that the tokens would be locked forever. But late that night, I had the sudden realization that if the tokens were still there, they could be recovered — by anyone

.

When anyone calls the burn

function on a Uniswap core contract, the contractmeasures its own liquidity token balance and burns it, giving the withdrawn tokens to the address specified by the caller. This is a core part of the intended behavior of Uniswap v2 (the basic mechanism is described in section 3.2 of the Uniswap v2 whitepaper).

I found the contract. The liquidity tokens were still there—and were worth around $12,000.

This meant three things:

- There was a ticking clock. Even if nobody else noticed the free money, anyone could remove their own liquidity at any time, accidentally receiving the tokens from the contract.
- I could put on my white hat and try to recover the tokens for the person who had accidentally sent them in. It would be as simple as calling the burn

function on the pool, passing it my own address.

- Except… I knew it wouldn't be simple.

The Dark Forest

It's no secret that the Ethereum blockchain is a highly adversarial environment. If a smart contract can be exploited for profit, it eventually will be. The frequency of new hacks indicates that some very smart people spend a lot of time examining contracts for vulnerabilities.

But this unforgiving environment pales in comparison to the mempool (the set of pending, unconfirmed transactions). If the chain itself is a battleground, the mempool is something worse: a dark forest.

The Dark Forest is my favorite science fiction book. It introduces the concept of a "dark forest" — an environment in which detection means certain death at the hands of advanced predators. In this environment, publicly identifying someone else's location is as good as directly destroying them. (This concept is also the inspiration for the Dark Forest game on the Ethereum testnet.)

In the Ethereum mempool, these apex predators take the form of "arbitrage bots." Arbitrage bots monitor pending transactions and attempt to exploit profitable opportunities created by them. No white hat knows more about these bots than Phil Daian, the smart contract researcher who, along with his colleagues, wrote the Flash Boys 2.0 paper and coined the term "miner extractable value" (MEV).

Phil once told me about a cosmic horror that he called a "generalized frontrunner." Arbitrage bots typically look for specific types of transactions in the mempool (such a DEX trade or an oracle update) and try to frontrun them according to a predetermined algorithm. Generalized frontrunners look for any

transaction that they could profitably frontrun by copying it and replacing addresses with their own. They can even execute the transaction and copy profitable internal transactions

generated by its execution trace.

That was why this rescue wasn't going to be simple. This burn

function could be called by anyone. If I submitted a transaction calling burn

, it would be like a flashing "free money" sign pointing directly at this profitable opportunity. If these monsters were really in the mempool, they would see, copy, mutate, and frontrun my transaction, taking the money before my transaction was included.

Note how much more brutal this environment is than even the Ethereum blockchain state itself. This free money had been sitting on-chain for about eight hours, undetected, waiting to be swept by anyone who called burn

. But any attempt to pick it up would get instantly sniped in-flight.

The Rescue

To try to extract the money without alerting the bots, I would need to obfuscate the transaction so that the call to the Uniswap pair couldn't be detected. This would involve writing and deploying custom contracts. Because I'm a professional DeFi thought leader, I had never actually deployed a contract to Ethereum before.

I needed help, and it was past midnight. Luckily, some of the best smart contract engineers I know live in a European time zone. My Paradigm colleague Georgios Konstantopoulos agreed to help with contract deployment and submitting the transactions. Alberto Cuesta Cañada, the lead engineer at another of our portfolio companies,Yield, volunteered to implement the contracts.

Some excellent Ethereum security engineers helped us come up with an obfuscation plan. In addition to burying the call as an internal transaction, we would split the transaction in two: a set

transaction that activates our contract, and a get

transaction that rescues the funds if the contract has been activated. This would be implemented as follows:

1. Deploy a Getter

contract which, when called by its owner, would make the burn

call ONLY if activated, and otherwise revert.

1. Deploy a Setter

contract which, when called by its owner, would activate the Getter

contract.

1. Submit the set

transaction and the get

transaction in the same block

.

The code for our custom smart contracts.

If the attacker only tried executing the get

transaction, it would revert without calling the burn

function. We were hoping that by the time the attacker had executed both the set

and get

transactions in sequence to spot the internal call to pool.burn

and frontrun us, our transactions would already be included.

To our surprise, the get

transaction would get rejected by Infura even when we manually overrode the gas estimator. After several failed attempts and resets, the time pressure got to us, and we got sloppy. We let the second transaction slip into a later block.

It was a fatal mistake.

Our get

transaction did get included—but with a UniswapV2: INSUFFICIENT_LIQUIDITY_BURNED

error, meaning the liquidity was gone. It turned out that, in the seconds after our get

transaction entered the mempool, someone had executed the call and swept the funds.

The monsters had devoured us.

The Lesson

Monsters are real

We knew intellectually that these generalized frontrunning bots existed. But until you've actually seen them in action, you're likely to underestimate them.

We held out some hope that doing the rescue as an internal call via an authorized contract, passing a variable from storage as the to

address, might protect us. It didn't.

If you find yourself in a situation like this, we suggest you reach out toScott Bigelow, a security researcher who has been studying this topic and has a prototype implementation for a better obfuscator.

Don't get sloppy

Even under time pressure, we should have stuck to the plan. If we had spent more time on the scripts, tweaked the contracts (perhaps changing the Getter

contract to do nothing instead of reverting if called before being activated), or even synced our own node to avoid using Infura, we probably would have been able to get the transactions into the same block.

Don't rely on normal infrastructure

The weirder what you're doing is, the harder it will be to jam it through existing infrastructure like Infura. In our case, we were trying to submit a transaction that looked like it would fail based on the current blockchain state, which Infura has reasonable protections against. Using our own node could have sidestepped this problem.

Better yet, if you happen to know a miner (we didn't), you could have them include the transaction directly in a block, skipping the mempool—and the monsters—entirely.

The future is only going to get scarier

This was just one example of a frontrunning incident. Similar things happen countless times every day. Today, the frontrunners are just bots. Tomorrow, it will be miners.

Miners today leave money on the table by not acting on these opportunities. In the future, they will

reorder and submit transactions in their mempools for their benefit. Even worse, they could reorg blocks mined by other miners, in an attempt to steal MEV which was not claimed by them, resulting in chain instability.

We think this future can be prevented, and are excited about several ambitious attempts to do so.Optimism has a vision of how MEV can be redirected for the benefit of the ecosystem, as part of its layer-2 scaling solution, optimistic rollup. StarkWare, in addition to its own layer-2 scaling systems, has built a “verifiable delay function” service called VeeDo that could make Ethereum applications immune to this kind of front-running attack. (Paradigm is invested in both Optimism and StarkWare.)

If you are thinking about MEV, or building something in this area, please reach out to us!