Flyweight: a DAPP to enable price-triggered orders for Uniswap v3 LP pools

January 2023
0xmn1
https://github.com/0xmn1

1. Abstract

Web3 users wishing to remain completely anonymous have limited options when it comes to placing ERC-20 price-triggered trade orders, without taking excessive product-specific risk. As of writing, the current most popular categories are DEXs (Uniswap, Cowswap, dYdX) & CEXs (Binance, Bitmex, Bybit), which expose users to excessive financial risk. Ideally, a product exists that does not subject these users to excessive risk, i.e.:

- Not give infinite token approval to a third-party
- Price-trigger orders that do not "reverse" if price retraces above the original price target
- Minimize financial risk in the event of a web3 hack/exploit/rug
- Not depend on a centralized entity vulnerable to changing user-impacting policies due to regulatory pressure
- Is always able to withdraw their money (i.e.: never exposed to "withdrawal pauses" from centralized entities or decentralized protocols)
- Can remain 100% anonymous (i.e.: no kyc, no identity document uploading, no email registration/verification, etc)

Therefore this white paper proposes a decentralized application design that solves all the above user problems with minimal tradeoffs.

2. Introduction

2.1 Uniswap

Uniswap v3, whilst having the most liquid decentralized pools currently out there, does not support "true" price-trigger orders. If one were to use Uniswap v3, they need to provide LP to a token pool with a configured *price range*. For example if the ETH/USDT pair currently trades at 1500, & a user wanted to place a price-triggered ETH buy order for 1300, they would have to add a LP position to ETH/USDT that only activates between 0-1300. Then they would get filled once ETH/USDT drops below 1300.

The issue with this is that it's not a *true* price-trigger order. After ETH/USDT drops below 1300, it can then re-trace back *above* their original price range of 1300, thereby "reversing" the order & the user ending up with an overly-weighted USDT LP position, which is the opposite of the user's goal.

2.2 Cowswap

This leads us onto Cowswap, a DEX that does actually support price-triggered orders. However the issue here is that like most similar products, they require infinite token approval to place price-triggered orders, & even to just keep the order open. For instance, to place an

ETH/USDT price-triggered trade order, the user has to first give infinite token approval to the Cowswap contract to spend an infinite amount of USDT in their wallet, as evidenced by the tx calldata containing a call to a function called "*approve*" with the max value of *uint256*.

Whilst all smart contract interactions have some risk attached, the user in this case is unnecessarily over-exposed in the event of a hack/exploit. It's possible that the user's wallet is entirely drained of all tokens with max approval, before they can react. The common problematic web3 user scenario is that once a hack/exploit is discovered, it's already happened & the user has to quickly revoke access (e.g.: via revoke.cash) & pray that they are not too late to do so.

2.3 dYdX

dYdX is another DEX, but requires you to deposit coins before opening price-trigger orders. The problem here is that it's common for traders to keep an allocation of their capital *liquid* in a ready-to-deploy format such as fiat/stablecoins, & another allocation for *active* trade positions such as ETH, FTT, SOL etc. Hence whilst dYdX is better from a risk-management perspective than Cowswap's infinite approval, the user's *liquid* capital allocation still remains unnecessarily over-exposed to risk.

2.4 CEX

This leads us to CEXs such as Bitmex, Mt gox, FTX, Binance, Bybit etc. Not long ago, these used to be trustworthy & liquid options, however most of them require KYC due to being under AML/CTF laws. KYC is usually not a user privacy issue, however the Celsius bankruptcy has revealed that *private* KYC info stored against creditors (users) actually gets made *public* due to US bankruptcy law, effectively doxxing addresses.

CEX users are also vulnerable to any hacks that leak KYC identification *documents*, which would subject the user to fraud, framing, telemarketing & other unethical/illegal activities.

Other points to mention include Binance's under-collateralized BUSD & proof-of-reserves liabilities transparency criticism, Bitmex's shutdown during the previous bear market nuke, & the Mt Gox hack. All of these have significantly deteriorated trust towards centralized entities. To top it all off, the recent FTX fraud has been the latest incident that has put the nail in the coffin for trading size on centralized entities, evidenced by the withdrawal of Wintermute funds off Binance & the reported increase in hardware wallet sales.

The FTX debacle has given "fuel to the fire" for regulators to introduce heavy crypto regulation, increasing risk to any crypto industry product that is vulnerable to regulatory pressure. Another user issue prominent in the FTX situation is the idea of "withdrawal pausing", where the centralized entity unexpectedly stops users withdrawing *their own money* in an unethical effort to keep the exchange liquid/alive.

With all the user risks identified, this paper will now move onto a proposed design that allows users to place price-triggered trade orders, without exposing them to the aforementioned risks.

3. A smart contract for storing anonymous orders

3.1 Persisting EOA address

A smart contract should be deployed that stores order data. For each order, only an ethereum EOA address should be requested/stored (i.e.: no KYC). The address (much like all states in this smart contract) should be publicly visible, in the best interest of transparency & therefore promoting trust between web3 users and the contract. The address should be persisted to:

- Enable a user to withdraw coins deposited for non-triggered orders at any moment (even in the event of a hack/exploit). Hence the common "*withdrawal pausing*" action sometimes taken by decentralized products should not become a future possibility. It goes without saying that this "trust logic" should be delegated to the smart contract (instead of a centralized entity). A check should be done at the contract-level to confirm the address as an EOA (e.g.: "*tx.origin == msg.sender*" in solidity) & to verify that the user has permission to cancel a specific order
- Once an order is executed, Immediately send resultant coins (from an executed order swap) to the order owner's address. A transfer instruction should be executed at the smart contract level at the earliest possible moment after trade execution, to transfer coin custody from the contract to the EOA. This is in contrast to other protocols such as Uniswap or Pancakeswap, **which retain custody until the user interacts with the contract** to exit/close a position & obtain custody.

3.2 Order state

Each order stored in the contract should have data to distinguish the order state (e.g.: a solidity enumeration), with the most important state being "executed", which prevents order reversal or duplicate swaps occurring.

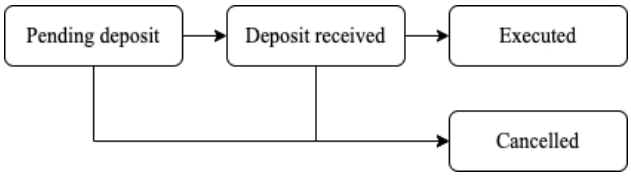

Figure A: state flow for orders

In a product comparison sense, this distinguishes it from the Uniswap price-range LP position where the initial swap can be "reversed" by price re-tracing.

3.3 Proxy contract design pattern

The contract's design **should not utilize** the "*proxy contract pattern*" or implement any sort of "*admin functionality*", even for the address that deployed the contract. This trades off developer upgradeability to promote user trust. This also rules out potential rugs such as when a proxy contract is upgraded, or when an admin address calls an admin function on a contract to unethically withdraw funds.

3.4 51% attack

Whilst a 51% **PoS** Ethereum attack is now extremely expensive & therefore unlikely, the design proposed by this whitepaper can be implemented for a contract on a vulnerable blockchain that is not ethereum (e.g.: a smaller PoS or PoW network), & thus should still assume a 51% attack as a realistic possibility & therefore the contract should be designed to remain operational in such an event.

It's important that the order state is updated from "pending deposit" to "deposit received" only *after* the on-chain transaction (that transfers coin custody to the contract to act as an order deposit) is confirmed. This means that if a 51% attack were to occur, & validators decide on a prior block to fork from, then the coins deposited to the smart contract still remain under contract custody & cannot be transferred to another address that is not the order owner. The scenario would look like this:

- Block 1 obtains sufficient validator confirmations
- Block 2: A benevolent user creates a new order in the contract
- Block 3: A malicious user also creates a new separate order, which will also be in a "*pending deposit*" state
- Block 4: The malicious user 51% attacks the network to update the contract state for their own order, from "*pending deposit*" to "*deposit received*"

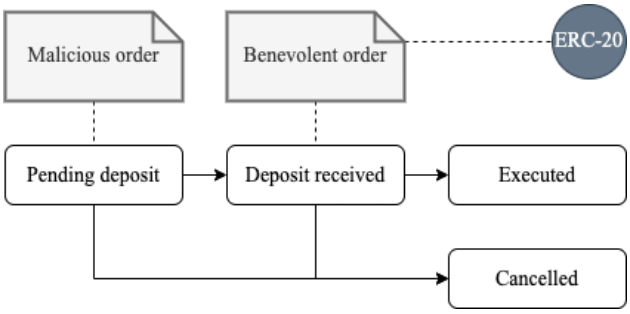

Figure B: block 4 order states

- Block 5: The benevolent user deposits the coins for their order, from their EOA to the smart contract

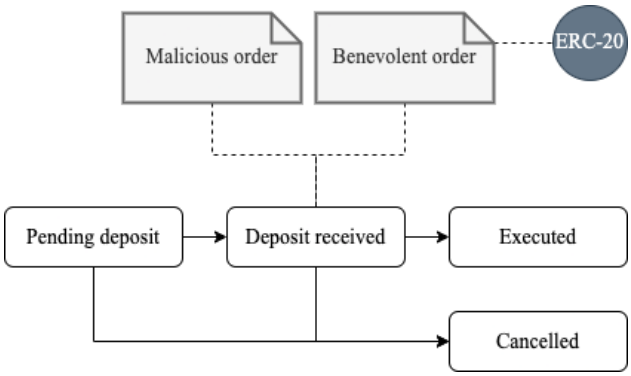

Figure C: block 5 sets the malicious order state to "deposit received"

- Block 6: New epoch occurs, 51% of validators agree to attestation
- Block 7: The malicious user subsequently adds a transaction to call the "*withdraw deposited coins*" contract function for their **own order**. Since the order state is "*deposit received*", custody for the benevolent user's coins (in block 5) is transferred from the contract to the malicious user's EOA

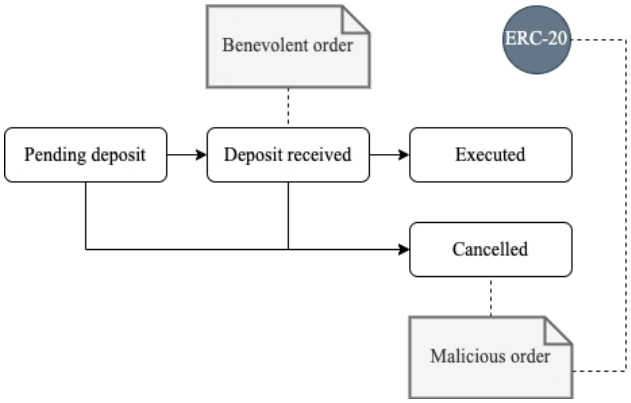

Figure D: block 7 transfers coin custody from the benevolent EOA to the malicious EOA

- Block 8: Benevolent user notices the hack (e.g.: through twitter), decides to execute the "*withdraw deposited coins*" for their **own order**. However the smart contract doesn't have enough coin balance & therefore an assert is thrown at the contract-level, failing the

transaction & burning remaining unused gas.

So far in this scenario, the legitimate user is out of luck & cannot withdraw their coins, because the custody has been transferred from the contract to the malicious user's EOA (even though the malicious user never deposited coins for their order).

Now let's continue the situation & assume the network regains control from the 51% attack (e.g.: validators disagree on prior epoch attestation & fork off a different block):

- Block 9: New epoch occurs, validators disagree to attestation in previous epoch & burn the malicious user's staked ETH, bringing the malicious user's staked ETH to < 51%
- Block 10: legitimate validators decide to fork off block 4
- Block 11: attestation in this epoch is agreed amongst majority validators

Hence, whilst the legitimate user's order state would have been reverted from "*deposit received*" to "*deposit pending*", they still have custody of the coins they initially deposited to the contract in block 5. This means the contract's balance remains correct/pure & the contract withdrawing functionality remains correct going forward.

4. A smart contract for whitelisting tokens

The smart contract should use the most liquid decentralized liquidity pools currently available, which are uniswap v3 LP pools.

When a token project rugs, the liquidity LP pool often gets "pulled" too. This introduces a new design problem, where swaps executed by Flyweight actually perform correctly, but due to the sudden lack of liquidity in an order's token pair, the Flyweight contract gets a bad swap quote from the Uniswap v3 router, but still executes it anyways. The end result: the smart contract user gets a less-than-expected amount of tokens from the automated swap. There are 2 alternate design options that can solve this problem:

1. Allow the user to configure a "*minimum slippage*" when creating/editing an order. This requires the smart contract to persist additional storage space (i.e.: higher gas cost).
2. Only support tokens that are likely to have liquid Uniswap v3 LP pools going forward. With this option, a **2nd smart contract can be deployed, containing a token whitelist**. The whitelist should contain both the token symbols & L1 token contract addresses (so that the addresses can be cross-verified by off-chain oracles).

Either design option can be considered viable, as they both achieve minimizing user financial risk.

5. Transaction verification oracle

For order states to update from "*pending deposit*" to "*deposit received*", an **oracle node** is required to verify on-chain transactions & connect ERC-20 transactions to orders. This **mechanism is what allows the design to not require infinite token approval**. This extra check does increase order creation time by 14 to 30 seconds (depending on ethereum validator confirmation time), but is often not noticed due to the Flyweight app "order creation" use case not involving quick successive interactivity.

Since the oracle needs to read on-chain transactions, one of the below options would need to be used:

- Running a blockchain client directly (e.g.: Geth if the design implementation is for Ethereum) & checking the block history
- Adding a dependency on a centralized service such as Infura, Alchemy or Etherscan.

To be completely transparent, this is a **centralisation dependency problem** that has not yet been solved in any app requiring oracle(s). Therefore ideally the "lesser of two evils" is chosen, which is to use the Etherscan API to retrieve on-chain data.

6. Trigger Oracle

In addition to the transaction verification oracle, another oracle is required to actually **trigger orders stored in the smart contract**. It needs a dependency to read token data (specifically last traded price & contract address), compare it against open orders in the contract (with state "*deposit received*"), then call a method on the oracle contract to:

1. Execute the Uniswap v3 swap
2. Update order status from "*deposit received*" to "*executed*"
3. Send resulting coins to order owner's EOA address

Again, to be transparent, this requires another centralized dependency on a token data provider, such as CoinMarketCap or CoinGecko's API.

7. Conclusion

This paper details a collection of smart contracts & oracles that can work together to manage price-triggered orders on a blockchain. But more importantly, provide an alternative product that serves web3 users which prioritize anonymity, custody, liability/risk minimisation & the evasion of token approvals. This design can be titled as "Flyweight", & when deployed collectively, it provides a unique alternative that solves real user problems in the crypto ecosystem.