Archibald Latham
May 31, 2023

OTC Swap Contract

For this exercise, I wrote and tested a contract that facilitates atomic OTC swaps of ERC20 tokens. Below I will highlight the design features of this contract as well as some relevant test cases.

Offer Struct:

```
struct Offer {
    bytes32 counterpartyRoot;
    address offerer;
    uint96 nonce;
    address offerToken;
    uint96 offerAmount;
    address considerationToken;
    uint96 considerationAmount;
    uint256 expiration;
}
```

In my approach, I define a struct called Offer that stores all of the relevant parameters for a single swap. One decision of note here is that as opposed to uint256, uint96 is used for the nonce, and token amounts. The advantage here is that the variables can be packed with the addresses for the offering party and corresponding token contracts respectively. This brings the struct size from 8 slots to 5 and improves gas costs. The downside of this decision is that the maximum value for the parameters is decreased, but considering the max value of uint96 is ~7.92e28 this is unlikely to be an issue for any reasonably defined token or even the most active OTC trader on human timescales.

Counterparty Merkle Root:

A requirement of this exercise was to consider that offering parties might want to restrict their counterparties to specific addresses. To confront this I implemented Merkle tree verification in the execution of offers. In the Offer struct users can define a Merkle root in one of three ways. If left empty it is assumed that any counterparty can execute the offer. In the case that there is specifically one counterparty, the offering party can simply set that address hash as the root. For multiple acceptable counterparties, the offerer can set a valid Merkle root that is verified on execution.

ECDSA Validation

A final consideration in this contract is the ECDSA signature scheme. For a user to offer a swap, they are not required to submit a transaction onchain and simply need to sign a message containing the parameters of the swap. This signature can then be passed offchain to the relevant counterparty to be verified in their execution of the swap. An advantage of this design is that the offering party can save significantly on gas costs as the only onchain actions they need to take are token approvals. A drawback of this design is that unsuspecting counterparties could face a griefing attack in attempting to execute invalid offers from a malicious offering party. Because this problem requires some gas cost from the attacker, in most cases can be solved by simulating transactions before execution, and does not lead to a direct loss of funds I would consider it a low severity or QA issue.

Testing

Finally, I tested my contract design using Forge. Due to our signature scheme, the contract is very minimalistic and only has two externally-facing functions, one to execute an offer given the Offer struct, signature, and Merkle proof and one to cancel offers by simply marking their nonce as executed. My test cases include: executing a valid offer, reverting execution when passed an invalid signature, reverting execution when passed an invalid Merkle proof, reverting execution when an offer has already been executed, reverting execution when an offer has passed expiration, and reverting execution when an offer has been canceled. I would consider the most relevant test cases here related to reverting execution when a nonce has been marked as executed. Ensuring that this piece of the contract is behaving correctly is crucial to preventing both replay attacks and reentrancy attacks.

Deployment

The Swap.sol contract is deployed and verified on Goerli and can be found here:
0x3dea1Aa84133fdd0f5F339a77d56b05F94299eBE

The contract and tests can be found in the repo below and run using Foundry:
https://github.com/0x0aa0/elthe