Parte 2

1)

To perform the attack send the 2 transactions to the network, spending the same bitcoin twice.

To be more efficient, the conflicts have not to happen in the ledger, but in the cache called MemPool, that every node has.

If the nodes receive conflicting transactions there, they discard them and keep those already accepted together with other non conflicting ones, forming in this way a candidate block that is eventually put in the final ledger.

After this building, follows the mining phase in which all miners compete and the transactions belonging to the MemPool of the winning node, will be the one to store. We cannot know previously if that was the car or sailboat acquisition.

In the case in which the 2 transactions are validated simultaneously by 2 different miners, generating 2 independent forks, will win the competition the one belonging to the longest chain, generally it is the one that has to wait for 6 confirmations before being accepted.

Another way to perform the attack could be by including both the transactions on the own chain of the attacker. But it is useless since other nodes could check the block and immediately detect it has no valid transactions and will simply reject it.

A way to succeed in this attack is by also using the 51% attack, so by having more hashing power in solving the PoW puzzles than the other miners.

As a matter of fact, if the attacker splits 2 different chains in which to put the 2 transactions and continues privately the one with the wrong transaction, so without broadcasting it to the rest of the network but where it still owns the bitcoins. Then, when this private chain becomes longer he can broadcast it and it will be accepted for sure.

2)

To detect the cheating, an anti cheating protocol is adopted, it uses some techniques for the building of the blocks such as the unconfirmed transaction, the multisig, the hash time locks and hash secrets.

The unconfirmed transactions are a way to collect locally some transactions and after a while publish them with the final balance on the blockchain, where they become confirmed.

To check if the new commitment is correct according to the old one already published are used scripts:

The P2MS Pay To Multisig Script associates transactions with multisig addresses, meaning that they are locked with m public keys and they require a subset of 1 or n private keys to be unlocked and to spend bitcoins.

In case of dispute of the channel capacity, we could introduce a third judicial party or do an escrow transaction.

Are necessary 2 signatures of the 3 parts, so it can happen that

- if both A and B sign the escrow then everything is ok since there is not a dispute

- if A refuses and B can provide the Proof of Shipment to C then B receives the bitcoins

- if B refuses and A can prove she has received nothing to C then A receives the bitcoins.

Another similar script is the P2HS Pay to Hash Script, in which the checking is done not with the keys but the hashing of the script: the commitment transaction is paired with a secret, the counterpart will know the hash of the secret but not the secret itself.

HTLC Hash Time Lock Contract is another way used to lock bitcoins using hash lock and time lock.

Hashlock for the preimage resistance means that before broadcasting the new commitment transaction and generating a new secret, reveal the previous secret.

Timelock instead means, that is represented by the image of a clock in the transaction of the output, is specified a time and a date in which can be verified an amount of blocks.

In conclusion, all these countermeasures are used with the aim to guarantee all the channel funds and there is a disincentive for the one cheating so both parties must have an interest in publishing the most updated balance. In case one party is discovered to cheat, all bitcoins of the channel go to the counterpart.

Parte 1

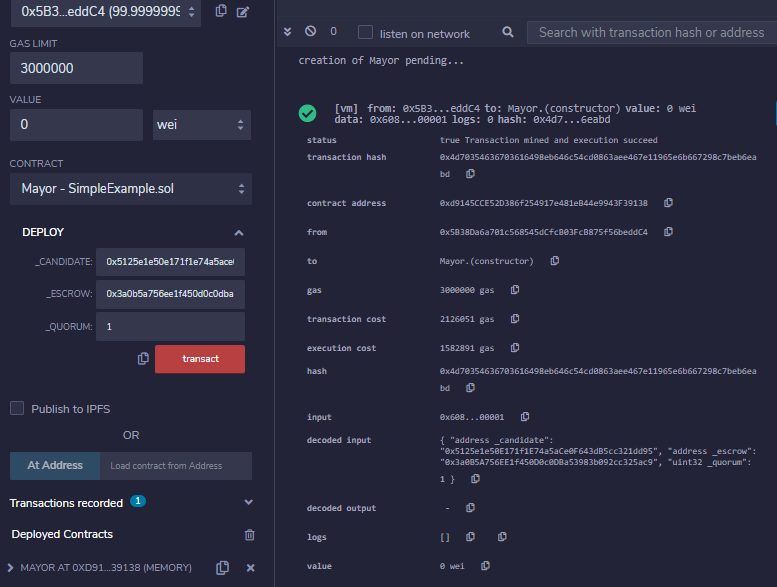
1) DanushiMayor.sol

2) For what concerns the cost in gas I plot some screenshots.

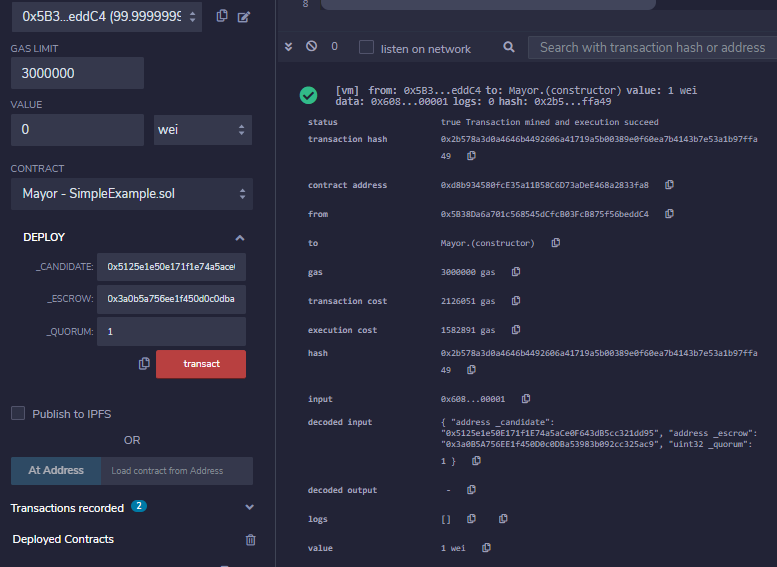
I first created some accounts such as

* the candidate = 0x5125e1e50e171f1e74a5ace0f643db5cc321dd95
* the escrow = 0x3a0b5a756ee1f450d0c0dba53983b092cc325ac9
* the voter entity = my address

Then I tried to deploy the contract by putting several values for the quorum input and the parameters of the other functions. This was useful also for debugging reasons since it shows step by step each part of the code execution.



When I tried to send 1 wei, it claimed that the amount was greater than how I could spend. This error was because the constructor was not with the payable clause. Adding that clause everything goes well and I get the following result:

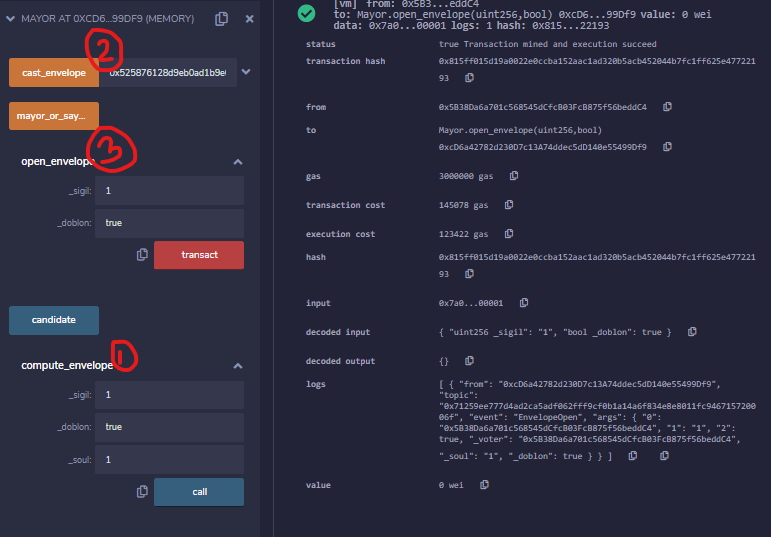


Now I wanted to achieve a situation where the candidate wins, so by putting yaySoul > naySoul.

If I immediately call the *mayor\_or\_cayonara()* function it complains that first I had to open all the sent envelopes, and so on.. so the check of the modifier functions are behaving correctly.

First I call the *compute\_envelop()* to get the envelope. Second, I call *cast\_envelop()* giving as input the previous output.

Third I call the *open\_envelope()*:

… and so on,

3) With what concerns security and possible vulnerabilities, I made the following considerations, adopting in some cases some countermeasures.

Within the function code we send ether and we know there are 3 possible ways for doing this. I first opted for the *address.transfer(value)* which is the most secure option but it limited the amount of gas to 2300 only. To avoid this limitation and also an improper use of it in enabling already known possible DAO attacks I changed it to the *address.call.value(value)(),* most recent version.

But as a pro of using *address.transfer(value)*, it could have helped for re-entrancy prevention. To solve this, after each ether transaction, the balance of the stored amount is updated to 0, in such a way that a recursive call of the function cannot resend again ethers before its older invocation.

For what concerns arithmetic operations, it is avoided the direct use of solidity math operations but are used instead the MathLibrary operations, which are more secure in checking for under and overflow issues, importing them from the github repository.

Since they are not prevented by those situations by which, such as putting an upper bound on the gas price or a commit reveal scheme through transaction sent with hidden information and check of the sent data after having put it on the block, the code of the function has the front running vulnerability.

I paid attention in never using the *transaction.origin* global variable but always used the *msg.sender*, then the phishing attack could be more difficult to happen.

Also *block.timestamp* is not used for randomness or escrowing funds or time dependent state change, so we are not a target for this type of manipulation attack.

4) It could be a target for attacks since anyone could see the application of the one way hash function and try to guess its result. For this reason, it must be kept secret not the function, anyone can know if we use keccak256 or another type of cryptographic SHA. The security of something does not depend on knowing which cryptographic tool we are using. It must be kept secret which parameters are used during its application. It should not have *uint sigil* as a parameter that could be guessed by someone trying to ‘observe’ the transaction while they are temporarily in the MemPool. It should depend on something more difficult to guess, for instance something random and related to the time or block number.