Trustology

Decoding an Ethereum Transaction.

It’s no secret, it’s just smart.

If you want to do anything interesting in Ethereum, you will have to interact with smart contracts. Whether you want to send ERC20 tokens like LINK or Dai, trade non fungible tokens like digital art, or earn interest on your crypto and interact with other DeFi products, a smart contract is always involved.

However, smart contracts are becoming increasingly complex. From proxy contracts to allow for upgradeability, to multi-send contracts that allow for the batching of transactions, what we are seeing is a rapid evolution of features that allow for the movement of one's assets.

DeFi transactions are often not generated by you or your code so you need to verify it's doing what you think it is. Additionally, understanding a method call is useful for:

- Seeing and knowing that the contract methods calls are what you expect
- Seeing the parameter types and values that allow you to understand how the contracts work
- Analysing a contract to produce stats on method calls
- Tracking interactions with key addresses
- Writing your own rules to decide on which transactions to sign (more on that later)
- Having fun, if you like that sort of thing

In part one of our new DeFi blog series, we will demonstrate and introduce what goes on inside an Ethereum transaction and how you can use Trustology’s custodial wallet platform to interact with smart contracts in a much more secure way. This knowledge will provide you with a good foundation in subsequent posts when we talk about our Firewall, Webhooks, and other DeFi services that we offer.

Let's dig in.

We’re going to start with a Gnosis Safe contract and a transaction a user would send that the Gnosis safe Dapp would generate.

Gnosis Safe is a popular wallet contract implemented by [Gnosis](https://gnosis.io). A multi-user organisation that wants to use Gnosis Safe will first have to define a list of accounts and a required threshold of signatures needed to send a transaction. Users will submit the transaction to the Safe, which will authorise the execution of the transaction only when the threshold required is reached. This way, the users have a tighter control of their funds.

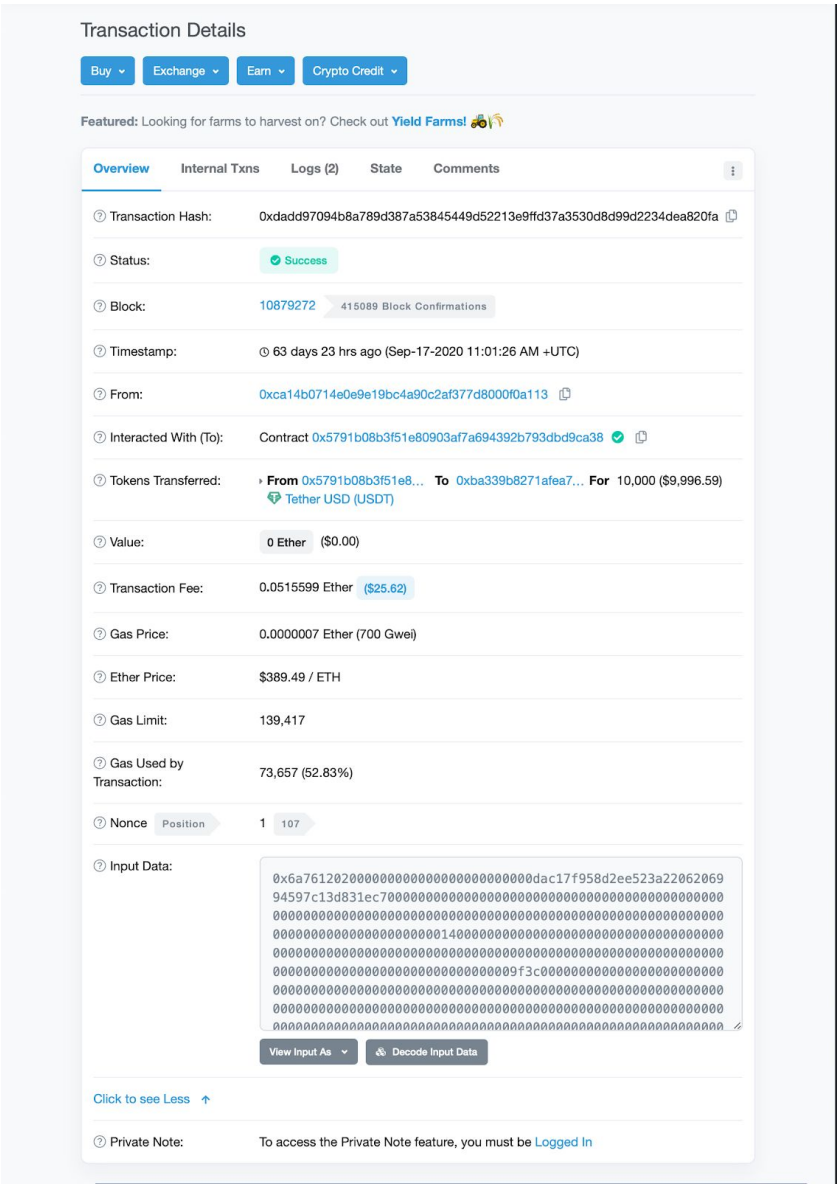
Trustology

The Gnosis safe has lots of features including offline signing (which saves on gas fees) but for now we're going to take a look at the basic method for executing a transaction.

For this blog we'll be looking at Etherscan data and seeing if we can get the data they do.

First port of call: Start with Etherscan

Let's start with a transaction as seen by etherscan.



The transaction above is from a user that has interacted with the Gnosis Safe smart contract on the Ethereum mainnet and is noted on Etherscan as:

Trustology

[transaction](https://etherscan.io/tx/0xdadd97094b8a789d387a53845449d52213e9ffd37a3530d8d99d2234dea820fa) .

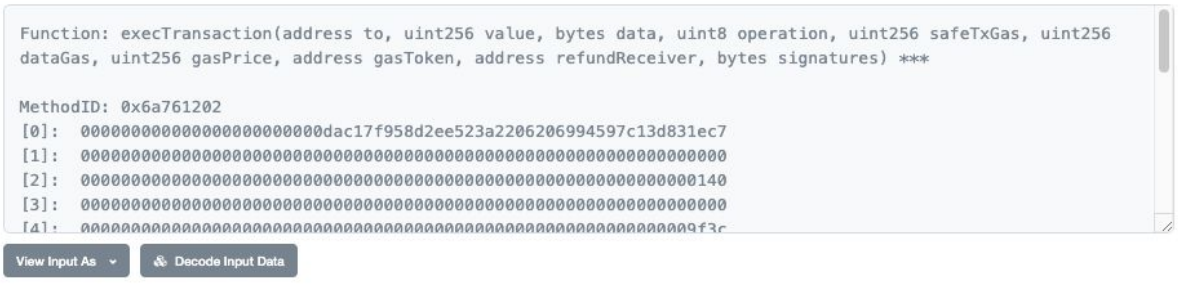
Let's start by looking at the "data" field. This is the payload that Ethereum uses to decide what method to call on the "to" (or "Interacted with") contract.

Here's the full data payload below. As you can see, a lot of bytes. This looks difficult to understand but with a little skill and knowledge we can break this up into the important parts.

```
`0x6a7612020000000000000000000000dac17f958d2ee523a2206206994597c13d83
1ec7000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000140000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000009f3c000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000001c0000000000000000000000000000000000000000000000000000000000000
00044a9059cbb0000000000000000000000ba339b8271afea713008314487dd98f8d9
41720f0000000000000000000000000000000000000000000000002540be4000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000c3827868e9e02b2d7557db27c
f7aef0b21e26a32b20b110c814f853c5948c6ba9a5b576241b59280334f04ce5bc385e4
2eaf1320b5099643a73089f847c42ec46f1b000000000000000000ca14b0714e0
e9e19bc4a90c2af377d8000f0a113000000000000000000000000000000000000000000
0000000000000000000001df862fe32b0bd3ff342134cb51570134168e1d81271ad44
9d9b3df8a24075d9b407496c6a4ce22265fc0b8f273ac5c129ba20070948d8cf5e2c619
df99aae0bf2000000000000000000000000000000000000000000000000000000000`
```

Start at the beginning

All "data" fields on an ethereum transaction contain the name of the method that is to be called. This is the first thing we should look at. Etherscan shows this for us in the "Input Data" section:



In this section we can see what Etherscan has given us:

```
Function: execTransaction(address to, uint256 value, bytes data, uint8
operation, uint256 safeTxGas, uint256 dataGas, uint256 gasPrice,
address gasToken, address refundReceiver, bytes signatures)
```

So, how did they get this?

Well, the first thing we need is to use the first 4 bytes (first 8 hex characters) of the input data, which is: 0x6a761202

This hex value is derived from taking the method name and its argument types, removing any whitespace, taking the `keccak` hash of the result, and then taking the first 4 bytes of it and displaying it in hex. With me so far?

> Note: The parameter names are not included in the hash. This means that different contracts can have the same methodId's but call their parameters differently and potentially have different logic. Thus it is sensible to keep track of the contract which the method belongs to. There is a handy website that tracks these. Check out https://www.4byte.directory/ and enter the hash 6a761202. At time of writing there is only 1 registered method.

We have the source of the Gnosis contract so we know what the method name and the parameters are. We grab them from here (https://github.com/gnosis/safe-contracts/blob/v1.2.0/contracts/GnosisSafe.sol#L114)

In JavaScript, the following will output the keccak hash of our method plus parameters:

```
// import a keccak decoder or write your own
import { keccak } from "../decoder/keccak";

const method = `execTransaction(address to, uint256 value, bytes data,
uint8 operation, uint256 safeTxGas, uint256 dataGas, uint256 gasPrice,
address gasToken, address refundReceiver, bytes signatures)`

// regex pattern to remove the word before a comma or closing bracket
export const removeArgsFromMethod = (method: string) => {
  return method.replace(/\s\w+(,|\))/g, (_, commaOrBracket) =>
commaOrBracket);
};

// remove the argument names and remove any spaces
const preparedMethod = removeArgsFromMethod(method).replace(/s/g, "")
// keccak hash of the method
const keccakHashOfMethod = keccak(Buffer.from(preparedMethod))
// first 4 bytes
```

```
const methodId = keccakHashOfMethod.slice(0,4).toString("hex") //
6a761202
```

Word of caution, If you don't have the source code then you won't be able to generate the hash to check.

The output from that Javascript is 6a761202 which matches the first 4 bytes (8 hex characters) of the data payload. Great. We've identified that the method being called is in fact "execTransaction" with the 10 parameters.

Now, if you go back to Etherscan and click on the `Decode Input Data` button in Etherscan you can see how Etherscan has taken the data payload and broken it up into the 10 parameters including the types. Again, how have they done this?

#	Name	Type	Data
0	to	address	0xdAC17F958D2ee523a2206206994597C13D831ec7
1	value	uint256	0
2	data	bytes	0xa9059cbb0000000000000000000000ba339b8271afea713008314487dd98f8d941720f000000000000000000000000000000000000000000000000000002540be400
3	operation	uint8	0
4	safeTxGas	uint256	40764
5	dataGas	uint256	0
6	gasPrice	uint256	0
7	gasToken	address	0x0000000000000000000000000000000000000000
8	refundReceiver	address	0x0000000000000000000000000000000000000000
9	signatures	bytes	0x827868e9e02b2d7557db27cf7aef0b21e26a32b20b110c814f853c5948c6ba9a5b576241b59280334f04ce5bc385e42eaf1320b5099643a73089f847c42ec46f1b0000000000000000000000ca14b0714e0e9e19bc4a90c2af377d8000f0a1130000000000000000000000000000000000000000000000000000000000001df862fe32b0bd3ff342134cb51570134168e1d81271ad449d9b3df8a24075d9b407496c6a4ce22265fc0b8f273ac5c129ba20070948d8cf5e2c619df99aae0bf20

Trustology

Get a handle on parameters

Our next job is to break up all the parameters that have been passed to the execTransaction and see what they are.

Before we do that a quick note on bytes to hex conversion

```
```math
1 \text{ byte of data} = 2 \text{ characters of Hex}
```
```

```
Here's why:
```math
1 \text{ byte} = 2^8 \text{ bits} = 256 \text{ bits}.
```
```

Hex has values 1,2,3,4,5,6,7,8,9,a,b,c,d,e,f. There are 16 of them. Thus two characters of hex will represent 16^2 = 256 bits. Hence two characters of Hex represent 1 byte of data. So whenever we talk about bytes, you can just double the number to get the equivalent number of hex characters.

Now, on to decoding the method parameters.

Let's have a look at the data again but this time we split it into 32 byte (64 Hex characters) strings. Why? Because ethereum core data size is 32bytes and nearly all primitive types are 32bytes. There are a couple of exceptions though.

So, once we strip off the methodId (0x6a761202) and then break the remaining data into 32 byte (or 64 character) chunks we can start to see something more interesting.

Index	Bytes	Notes
0	000000000000000000000000dac17f958d2ee523a2206206994597c13d831ec7	to
1	0000000000000000000000000000000000000000000000000000000000000000	value
2	0000000000000000000000000000000000000000000000000000000000000140	v data (encoded later)
3	0000000000000000000000000000000000000000000000000000000000000000	operation
4	0000000000000000000000000000000000000000000000000000000000009f3c	safeTxGas
5	0000000000000000000000000000000000000000000000000000000000000000	gasPrice

6	0000000000000000000000000000000000000000000000000000000000000000	gasToken
7	0000000000000000000000000000000000000000000000000000000000000000	refundReceiver
8	0000000000000000000000000000000000000000000000000000000000000000	dataGas
9	0000000000000000000000000000000000000000000000000000000000000001c0	v signatures
10	0000000000000000000000000000000000000000000000000000000000000044	data (encoded later)
11	a9059cbb000000000000000000000000ba339b8271afea713008314487dd98f8	
12	d941720f0000000000000000000000000000000000000000000000000000000002	
13	540be400000000000000000000000000000000000000000000000000000000000	
14	00000000000000000000000000000000000000000000000000000000000000c3	
15	827868e9e02b2d7557db27cf7aef0b21e26a32b20b110c814f853c5948c6ba9a	signatures
16	5b576241b59280334f04ce5bc385e42eaf1320b5099643a73089f847c42ec46f	
17	1b000000000000000000000000ca14b0714e0e9e19bc4a90c2af377d8000f0a1	
18	1300000000000000000000000000000000000000000000000000000000000000	
19	0001df862fe32b0bd3ff342134cb51570134168e1d81271ad449d9b3df8a2407	
20	5d9b407496c6a4ce22265fc0b8f273ac5c129ba20070948d8cf5e2c619df99aa	
21	e0bf200000000000000000000000000000000000000000000000000000000000	

These are the values of the input parameters of the `execTransaction` function. It's not quite straightforward so lets work through each 32 byte value in turn. Column 3 of the table above shows where each of the values maps to the parameters of the execTransaction function.

For example, the first parameter (the `to` parameter)

`000000000000000000000000dac17f958d2ee523a2206206994597c13d831ec7`

is of type `address`.

Ethereum addresses are 20 byte values so it has been zero padded to fit into a 32 byte value. To get the actual address, we just need to extract the last 20 bytes and prefix 0x.

In this case it becomes: `0xdac17f958d2ee523a2206206994597c13d831ec7`. (Hmm...this is the Tether contract)

In fact, removing the zero padding is how to decode most of the simple parameters like `uint8`, `unit256`, etc...

The next parameter is the “value”. This, we assume, is how much value needs to be transferred from the Gnosis safe. It’s value is zero.

The next parameter is interesting, it's of types bytes. Bytes is a variable length, so how long is it? The value is:

`0000000000000000000000000000000000000000000000000000000000000140`

This value represents the offset in hex of bytes from the `methodId` of where the actual data begins. Think of it as a pointer, redirecting you to where you should look for this information. So we need to convert 0x140 bytes into hex characters.

```math
0\text{x}140 \text{ (bytes in hex)} = 320 \text{ bytes in decimal}. \\
\quad \\
320 \text{ bytes in decimal} \times 2 = 640 \text{ characters in Hex}.
```

so the data begins 320 bytes from the start, and thus it is 640 hex characters along, or 10 rows along. In JavaScript, it will look as follows:

```ts
const numberOfBytesInHex = 140
const numberOfHexCharacters = parseInt(numberOfBytesInHex, 16) * 2; // = 640
```

Going to the 10th row the first 64 characters represent the length of the data (this requires the same conversion done above).

`0000000000000000000000000000000000000000000000000000000000000044`

is decoded into a length of 136 hex characters.

So finally on the 11th row the data begins and ends after 136 characters. Which means our data field is the following:

```
`a9059cbb0000000000000000000000000ba339b8271afea713008314487dd98f8d941720f
0000000000000000000000000000000000000000000000000000002540be400`
```

So in summary, for a bytes field, the first 32bytes is a pointer to where the data starts, once there, the next 32bytes tells you how long the data is. Then you read that many bytes. If the data doesn't end in a 32byte block, it will be padded so whatever is next, will start in the next 32byte block.

Looking at the value again:

```
a9059cbb0000000000000000000000000ba339b8271afea713008314487dd98f8d941720f0
0000000000000000000000000000000000000000000000000000002540be400
```

It looks a little familiar? It has a 4 byte methodId and then 2 32 byte chunks (2 lots of 64 hex characters). It's another method call!

In fact, we can recognise immediately that this transaction is in itself an ERC20 transfer transaction as it has the famous `a9059cbb` method signature.

(Tip: go to https://emn178.github.io/online-tools/keccak_256.html and type in `transfer(address,uint256)` and you will see the first 8 characters are in fact `a9059cbb`)

So, we can see that this Gnosis transaction is to call another contract of address `0xdAC17F958D2ee523a2206206994597C13D831ec7` with the "data" of:

```
a9059cbb0000000000000000000000000ba339b8271afea713008314487dd98f8d941720f0
0000000000000000000000000000000000000000000000000000002540be400
```

Which is an erc20 "transfer" call. We can then use what we learnt above to recursively calculate the details of this data call.

Take a look and see if you can work out the recipient and value of tokens sent to the Tether contract?

Conclusion

There was a lot of technical information here but it was very useful to be able to break the transaction data down and by working through a data payload methodically you can always understand what's happening.

We took a look at a Gnosis contract and broke it down into the constituent parts. We saw the methodID and the parameter types and how to deconstruct them into useful arguments.

Trustology

We also saw how that, in this case, there was effectively a transaction within a transaction. The Gnosis safe transaction was just wrapping an ERC20 transfer. This was effectively moving USDT tokens from the Gnosis safe to somewhere else.

In our next blog, we'll see how we can take our understanding of Ethereum transactions to break down a Defi contract call to apply rules to decide if we want to sign / send the transaction. This is something that Trustology uses in its new Defi firewall to protect customers from signing / sending transactions that don't meet certain criteria. The DeFi Firewall is the first in a suite of services Trustology aims to launch for institutional users looking to support new tokens in DeFi or explore yield bearing opportunities. Notifications and flows are next on our list to complement our current firewall solution. Currently, we are the only custodial wallet to integrate with MetaMask, which lets institutions tap DeFi innovation from the security of an insured custodial wallet platform.

About Trustology

Backed by ConsenSys and Two Sigma Ventures, and founded in 2017, Trustology was established on the premise of enabling greater freedom to transact in a fair and efficient manner. By bringing the world of blockchain technology and cryptoassets to new market participants, we believe this will help make this happen.

That's why we built TrustVault — a fast, user-friendly, insured and highly-secure custodial wallet service for institutions and individuals designed to address the security and ownership shortcomings of existing solutions today. Learn more about us at trustology.io.

Recommended reading:

If you want to know more about Ethereum transactions there are some good articles, you start with:

- Understanding an Ethereum Transaction
- The transaction payload