A painless guide to working with the Ethereum network

Published on Dec 18, 2019. Written by Niklas Baumstark (https://twitter.com/_niklasb).

Amongst the core features of the Cashlink product (https://cashlink.de/en/homepage-en/) is a smart contract that represents digital securities on the Ethereum network. Approximately one year ago, we started working on this smart contract component as well as the surrounding infrastructure, forming the basis for a convenient user interface to manage and transfer digital securities.

During 2019 we have deployed smart contracts for several of our customers and integrated them into our security issuance process. We learned on this journey that working with the technology is far from straightforward – high-quality resources about deploying Ethereum infrastructure and interacting with it in production is scarce. In this blog post we want to give other developers working with Ethereum insights into our process for designing and developing smart contracts. We will also give an overview of the tooling we built to manage smart contracts deployments in production and work with the network in a secure and robust way.

We will put a particular focus on the technical challenges that we faced at Cashlink, our considerations about the available design space in these areas, and our approach to tackling them:

- How to design & implement smart contracts, and debug them on the mainnet if things go wrong;
- How to interact with the network in a robust way to ensure smooth operations and handle failures gracefully in a production environment;
- How to manage identity in a cloud environment.

Smart contract design & development

This is the aspect of the technology which most of the existing resources focus on, with mature tools readily available, such as Remix (https://remix.ethereum.org/), Truffle (https://www.trufflesuite.com/) / Ganache (https://www.trufflesuite.com/ganache) and compilers (https://solidity.readthedocs.io/) targeting (https://vyper.readthedocs.io/) the Ethereum virtual machine (EVM). When in doubt, we found that the authoritative Ethereum yellow paper (https://ethereum.github.io/yellowpaper/paper.pdf) is a rather accessible source on the low-level details of the protocol and virtual machine, and the Ethereum wiki (https://github.com/ethereum/wiki/wiki/JSON-RPC) is great for looking up details of the JSON-RPC API provided by nodes.

At Cashlink, like probably almost anywhere else, we use the Solidity language to implement our smart contracts, and the Truffle suite for unit testing. In a future blog post, we will go into more details about the design goals and implementation of our digital security smart contract, but here

is a quick preview of two properties we have identified to be particularly useful while running smart contracts in production:

- Feeless operability – Our philisophy is that really good user experience can only be achieved by abstracting away the complexity imposed by the actual Ether currency. Thus, all of our operations can be executed via a cryptographic claim signed by a participant off-chain, rather than a signed Ethereum transaction from that participant. This allows us to separate the orthogonal concerns of authorization (https://en.wikipedia.org/wiki/Authorization) and transaction settlement, which are often interleaved by using sender identity for access control. More on that later.
- Upgradeability – If our experience in application security has taught us one thing, it is that we can never be 100% certain of the absence of security-related bugs in any software, including smart contracts. Even with correct source code, compiler bugs (https://solidity.readthedocs.io/en/latest/bugs.html) might introduce vulnerabilities. Our stance is that security updates should be possible for any non-trivial smart contract.

Debugging and testing on the mainnet

Clearly, before running a contract on the mainnet, it should be deployed and tested on a development network – we recommend a local node, or alternatively the Goerli network (https://github.com/goerli) due to its reasonably small size and high compatibility. However, debugging and testing on the mainnet is sometimes necessary to analyze production issue. A very useful and underappreciated tool for this is the forking mode of Ganache (https://github.com/trufflesuite/ganache-cli#options): provided with an upstream Ethereum node and a block number, Ganache can act as a local simulation of the upstream network at the given point in time. This allows us to issue arbitrary transactions without affecting the state of the real network, and even impersonate wallets without access to the corresponding private key.

This yields a straightforward method to test an existing smart contract's functionality: We can fork the mainnet at the block, and simply issue test transactions to ensure the contract behaves as expected.

Another use case is debugging a specific transaction such as a smart contract call: we can fork the network right before the block where the transaction occurred, unlock the sender wallet and use

```
eth_sendRawTransaction
```

(https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_sendrawtransaction) to replay it verbatim. If you are interested, we have prepared a simple example script (https://gist.github.com/niklasb/141a82a8602aa85aad1dc4b101df0602) to replay a transaction on the rinkeby network, using Ganache and Python with Web3.py (https://web3py.readthedocs.io/en/stable/). We can also see there that the

```
debug_traceTransaction
```

API (https://github.com/ethereum/go-ethereum/wiki/Management-APIs#debug_tracetransaction) works as expected, so

```
truffle debug
```

(https://www.trufflesuite.com/docs/truffle/getting-started/debugging-your-contracts#command) can be used for source-level debugging.

We should point out that this is not a perfect solution: There have (https://github.com/trufflesuite/ganache-core/issues/474) been (https://github.com/trufflesuite/ganache-core/issues/446) several (https://github.com/trufflesuite/ganache-core/issues/436) correctness (https://github.com/trufflesuite/ganache-core/issues/494) issues (https://github.com/trufflesuite/ganache-core/issues?utf8=%E2%9C%93&q=is%3Aissue+label%3Aforking+) with Ganache's fork implementation in the past, but the ones we ran into in production have been fixed in recent months – presumably because people have been using this feature more.

Interacting with the Ethereum network

No matter what automated infrastructure we plan to implement on top of Ethereum – be it for deploying smart contracts, funding wallets with ETH, making smart contract calls, reading out information such as token balances, or listening to Solidity events – we need a solid and robust way to "talk to the network". In our case, we need to do all of the above: Our smart contracts heavily rely on events to output status updates for our business code.

There are at least two prominent modes of accessing the network, each with their own advantages and disadvantages:

- Self-hosted node – Here we run an Ethereum node such as Geth (https://geth.ethereum.org/) or Parity (https://www.parity.io/ethereum/) on dedicated hardware. The advantage of this method is that no trust has to be put into additional third parties. This comes with the downside of having to manage the node instance on-premise, which can be particularly painful due to the large size that the Ethereum blockchain has accumulated over time. To give some specific numbers, as of November 2019 when this post was written, a "fast mode" sync of the chain to a hard disk (not SSD), using Geth on a machine with 12 cores, took almost a week to complete and occupied almost 500 GB of disk space. Syncing a full archive node with Geth seems almost impossible (https://blog.slock.it/how-to-not-run-an-ethereum-archive-node-a-journey-d038b4da398b) without a large array of SSDs and a lot of time at hand. In addition, as with any other self-hosted solution, maintaining uptime is hard. At times, we also observed periods of 20 minutes or more where our node would not be able to sync with the rest of the network, therefore representing an old state, a problem which we have not fully investigated yet.
- Third party-hosted node – The most prominent service provider in this area is Infura (https://infura.io/), which provides features that are hard to achieve using self-hosting, in particular full archive data (https://infura.io/docs/ethereum/add-ons/archiveData) and an index for EVM logs (https://blog.infura.io/faster-logs-and-events-e43e2fa13773), the mechanism used to implement Solidity events

(https://solidity.readthedocs.io/en/latest/contracts.html#events), which supports enumeration of logs in a given block range. The latter is particularly important for applications processing the output of a smart contract, which is typically realized via events. The canonical example are the

`Transfer`

events in the ERC-20 token interface (https://eips.ethereum.org/EIPS/eip-20), logging token transfers including meta information such as sender, receiver and amount. As of the last time we checked, one limitation of the Infura API was the lack of the

`debug_*`

(https://github.com/ethereum/go-ethereum/wiki/Management-APIs#debug) API, which can be used to get a full execution trace of a transaction for debugging purposes. In order for this to be supported on a self-hosted node, full archive data is required. For completeness we want to mention that alternative node providers such as Alchemy (https://alchemyapi.io/) exist, which we have not yet had the chance to evaluate.

As mentioned above, we critically rely on log output of our smart contracts. Without a third-party log index, we would have to rely on the slow

`eth_getLogs`

(https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_getlogs) implementation of Geth – which can take minutes to enumerate the logs of a single smart contract on a mainnet – or set up our own indexer based on filters (https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_newfilter). However, overly relying on and trusting solely in a third party appeared to us to be a risky endeavour. We recommend a hybrid approach that combines the best of both worlds: For reading from and querying the network, use a third-party node and cross-check the resulting information with a self-hosted node if possible. Querying by block number is always cheap in Geth. For dry-running (https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_estimategas) and eventually broadcasting transactions (https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_sendrawtransaction), we recommend using a self-hosted node, and cross-checking the correct mining of transactions via a third party.

Leaky transaction abstractions

When consulting basic blog posts and tutorials about working with Ethereum smart contracts, we could arrive at the conclusion that the following is a best-practice process:

1. Use some Web3-like library such as web3.js (https://web3js.readthedocs.io/en/latest/) or Web3.py (https://web3py.readthedocs.io/en/latest/), and connect it to an Infura node;
2. provide the library with a wallet's ECDSA private key for transaction signing – a concept called account unlocking;
3. create a contract (https://web3js.readthedocs.io/en/v1.2.4/web3-eth-contract.html) object based on the contract ABI (https://solidity.readthedocs.io/en/latest/abi-spec.html);

4. use a single API (https://web3js.readthedocs.io/en/v1.2.4/web3-eth-contract.html#methods-mymethod-send) to assemble a smart contract call transaction and "send" it to the network with a dynamically configured gas price;
5. wait for the receipt of the transaction.

While this looks like a very simple and easy to use abstraction for the concept of "calling a smart contract function", it conflates very distinct aspects of that concept into one unified abstraction, which renders it in fact rather leaky (https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/):

- Transaction assembly – the process of deriving the payload of the transaction from the intended smart contract call, in particular the

  `data`

  field of the transaction.
- Identity management – the management of private keys, to be able to sign assembled transactions (and potentially other, application-specific messages) transparently and on demand.
- Transaction execution – the strategy by which a gas price is selected, the transaction is broadcasted to the network, and completion of the transaction is determined.

One example where this leakiness becomes apparent is that of starving transactions: Imagine that the average gas price in the latest mined block is currently x. As a conservative estimate, in step 4 from above we might configure a gas price of $1.2 \cdot x$ (which is by the way more than what web3.js would configure by default) and broadcast the transaction, waiting for the receipt. But what if, by some unlucky coincidence, suddenly the transaction load of the network increases for multiple hours or days and miners now only consider transactions with gas price of at least $2 \cdot x$. In order to handle this scenario, we first of all have to detect it and repeat steps 4 and 5, carefully constructing the transaction such that it has the exact same parameters, including the same nonce as the previously broadcasted transaction. Suddenly we have to ensure deterministic transaction assembly, and also manage the nonce manually, the latter of which is an aspect that the API was deliberately trying to hide from us for simplicity reasons.

Another example is transaction confirmation in the presence of ephemeral network forks, also known under the fancier name chain reorganizations (https://blog.ethereum.org/2015/08/08/chain-reorganisation-depth-expectations/): To have high confidence that a transaction will stay in the network for good, it is not sufficient to just wait for the transaction receipt from the upstream node. Rather we have to wait for a rather large number of suceeding blocks to be mined and announced to gain this confidence, which requires an explicit polling strategy. The abstraction of synchronously sending a transaction and waiting for it to complete is thus also leaky.

Of course the JSON-RPC Ethereum node API provides access to slightly more low-level primitives which allow us to be more explicit, we just have to be mindful of these considerations and set up our interactions at the appropriate level of abstraction.

When designing our smart contracts and the infrastructure we use to connect them to our business logic, we set out to make these responsibilities explicit and strived for the following design goals:

1. Read access to the network is much easier and less security-critical than write access, due to the fact that no identity management is required. Thus it can and should be integrated into the normal business logic beneath just a thin layer of abstraction (for example to provide caching and the cross-checking logic described above).
2. Identity management should be clearly separated from the transaction execution logic.
3. Transaction execution should have direct access to the transaction assembly logic, so that the same transaction (modulo gas price) can be broadcasted with increasing gas prices, and speedy completion can be ensured.

You might wonder how identity management and transaction assembly/execution can be separate concerns, when it is required to sign transactions after assembly. Our solution to this is based on the feeless operability of our smart contract, where the identity of the sender (the

`from`

field of the transaction) can be distinct from the identity that authorizes an action. In fact every single one of our transaction is signed with a completely exchangeable worker wallet with the only purpose of paying the transaction gas fees. The entire transaction execution engine is designed as a separate service from the rest of our business logic, and runs on a separate system in production. Signed claims by the actual authoritative private keys are provided to this service as parameters, and are incorporated in the transaction assembly phase.

An overview of useful low-level abstractions

In this section we will quickly review the APIs we use in our transaction service to implement the logic outlined above. Since our stack is based on Python, we will use Web3.py (https://web3py.readthedocs.io/en/latest/) as an example, but similar functionality also exists within web3.js or its dependencies.

- For Solidity parameter encoding,

  `eth_abi.encode_abi`

  (https://eth-abi.readthedocs.io/en/latest/encoding.html#encoding-python-values) is used as a low-level primitive, used for constructing custom packed object representations. If the ABI description of a Solidity contract is available,

  `web3.contract.Contract.encodeABI`

  (https://web3py.readthedocs.io/en/latest/contracts.html#web3.contract.Contract.encodeABI) provides a convenient wrapper to encode the call data for a method call with a set of given arguments.
- Common types that represent ECDSA signatures, private and public keys can be found in

  `eth_keys.datatypes`

(https://github.com/ethereum/eth-keys). This library in general is highly recommended since it is well-designed and provides canonical abstractions for most of the Web3.py (https://web3py.readthedocs.io/en/latest/) ecosystem, as well as low-level functionality such as ECDSA signature primitives.

- Once a transaction is assembled,

  ```
  web3.eth.estimateGas
  ```

  (https://web3py.readthedocs.io/en/latest/web3.eth.html#web3.eth.Eth.estimateGas) is a useful tool to dry-run the transaction to determine the approximate gas usage, and to avoid broadcasting it in case it cannot succeed in the first place.
- For deriving keys using the popular hierarchical deterministic wallet (https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki) standard, the ethereum-mnemonic-utils (https://github.com/vergl4s/ethereum-mnemonic-utils/blob/fe222052dc800a4e02fd98b48f5e4b106f9733f4/mnemonic_utils.py) code forms a good basis.
- Last but absolutely not least, py-evm (https://py-evm.readthedocs.io/en/latest/) and ethtester (https://github.com/ethereum/eth-tester) is a great combination of tools for testing the interactions with smart contracts, with a similar feature set to Ganache from the JavaScript world. We use it for full integration tests of our transaction execution engine with our smart contracts.

Identity management

The simplest form of identity management in Ethereum is what can be described as *sender-based identity*. The identity associated with a smart contract action is equal to the sender of the transaction, and authorization (https://en.wikipedia.org/wiki/Authorization) is performed against this identity. We outlined above how this form of authentication leads to an entanglement of identity and transaction execution logic which can be undesirable.

To give an example, consider a token smart contract with an

```
issue(recipient, amount)
```

function, which is only supposed to allow transfers if the action is authorized by the identity representing the issuer of the token. There are at least two alternative approaches to sender-based identity which can be used to implement this.

- On-chain identity – This describes an on-chain representation of identity in the form of a smart contract via protocols such as ERC-725 (https://github.com/ethereum/EIPs/issues/725) and ERC-735 (https://github.com/ethereum/EIPs/issues/725)/ERC-780 (https://github.com/ethereum/EIPs/issues/780). Different identities (such as a "token issuer" or a "token holder") are represented using distinct smart contracts. Network participants – including the owner of the identity themselves – can add signed cryptographic claims about an identity. This allows for the separation of the identity as a concept, and a specific Ethereum wallet. In our example, with a simplified implementation, the issuer could be represented by such a smart contract, and they could add a claim about themselves such as

"Yes, I want to issue 100 tokens to wallet X". The token smart contract could then look up this claim and authorize a certain issuance action based on it. The problem here is that in order to manage the identity, smart contract interactions with the identity contract are required, and thus the problem of separating identity management and transaction execution persists.

- Off-chain identity – With this approach, an identity is still represented as an Ethereum wallet, however smart contract methods are augmented by a parameter that provides a claim signed by that identity. In our example the

  ```
  issue
  ```

  function could be changed to

  ```
  issue(recipient, amount, issuerClaim)
  ```

  . This is a basic example of a feeless smart contract design. We can add more flexibility to this design by making use of the chain of trust (https://en.wikipedia.org/wiki/Chain_of_trust) principle: the *root of trust* Ethereum wallet (let's call it X) of an identity can sign a claim of the form *"Me, wallet X, confirms that wallet Y may authorize issuance actions in my name"*, and this claim concatenated with the claim *"Me, wallet Y, confirm that 100 tokens should be issued to Z"* would be just as valid as the latter claim signed by X. Combined with appropriate protections against replay attacks, and a simple on-chain registry for claim revocation (https://en.wikipedia.org/wiki/Key_(cryptography)#Ownership_and_revocation), this allows us to manage an identity almost fully off-chain.

The latter approach is what we use in our systems, and we plan to detail the smart contract implementation in more depth in a future blog post. The important take-away in terms of infrastructure is that we can fully separate the identity management (in this case, the signing of various claims) and transaction execution logic in our systems, which gives us maximum flexibility and allows for strong isolation of the root key material.

Maintaining the root keys

No matter what method we choose to represent identity in our systems, the root of trust is always effectively formed by one or multiple ECDSA private keys, and key management becomes a fundamental challenge. The main choice is between using an offline storage mechanism, such as a paper or hardware wallet (referred to as cold storage in the Bitcoin universe), or an online storage, such as a private key in main memory or a dedicated hardware security module (HSM) (https://en.wikipedia.org/wiki/Hardware_security_module).

Within the threat model (https://en.wikipedia.org/wiki/Threat_model) of our off-chain identity management, an attacker having one-time access to a ECDSA signing oracle is not much different from them having permanent access to the actual private key, although this might differ for other identity management approaches. When considering whether an HSM is useful under a certain threat model, we should take into account that the capabilities required to extract a private key from main memory are similar to those required to gain one-time access to the HSM as a signing oracle – the most likely attack scenarios being privileged native code execution on, or physical access to the machine which is interacting with the key.

A defense in depth (https://en.wikipedia.org/wiki/Defense_in_depth_(computing)) approach could use the following strategies:

- Different root keys are used wherever possible: For example, every identity should have a different root of trust. Even if some identities such as "KYC provider" and "token issuer" are represented by the same legal entity, there is no reason other than convenience to interleave these identities by using the same root key. It helps to make use of hierarchical deterministic wallet (https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki) to derive multiple keys from a single seed.
- A physical separation exists between long-lived root keys, and shorter-lived intermediate keys. At Cashlink we do this by using a hardware wallet as our long-lived root key which is only ever connected to a single Chromebook device that is only used for the purpose of creating claims that transfer authority to intermediate keys (with a fixed expiration date). This dedicated device plays the role of the *high assurance workstation* as recommended by Trail of Bits (https://blog.trailofbits.com/2018/11/27/10-rules-for-the-secure-use-of-cryptocurrency-hardware-wallets/).
- Short expiration times for intermediate keys, and regular rotation of keys.
- In the larger software architecture, identity management is separated from the rest of the system, including transaction execution.

When these strategies are strictly implemented, important keys are exposed to minimal attack surface, and the impact of infrastructure compromises is minimal in the sense that the keys stored on them – or accessible by them via signing oracles in the case of HSM – have the least possible privilege and lifetime to ensure unsupervised operations.

Overall, key management is a very hard problem and here we just outlined one possible solution to deal with this complexity. We are always curious to learn how other participants in the Ethereum, and of course also the wider blockchain ecosystem handle this challenge.

Wrap-up

To conclude, we hope that we could lay out some of the maybe not immediately obvious challenges with running Ethereum-based applications in production. The key take-aways for us can be summarized as follows:

1. Carefully design your smart contracts with automated usage and surrounding infrastructure in mind.
2. Consider moving identity management off-chain where applicable.
3. Separate out transaction execution logic, possibly into dedicated services/machines, to achieve flexibility, high reliability and safety.
4. Consider using the lower level abstractions provided by the frameworks and tools at hand, such as web3.js and Web3.py (https://web3py.readthedocs.io/en/latest/), to gain back control over various aspects of how you interact with the Ethereum network.
5. Combine third-party node providers with your own Ethereum node(s) for fast queries in addition to high availability and confidence.

6. When automatically deploying new smart contracts, consider using the "deployer contract" design pattern.

As a very opiniated post-script, we are a big fan of the combination of Python and Ethereum, especially thanks to the excellent library ecosystem. Nevertheless, Truffle still remains the best way to efficiently develop Solidity smart contracts for us.

Where can I learn more?

- The Ethereum wiki (https://github.com/ethereum/wiki/wiki) is the official documentation of the Ethereum protocol and EVM.
- BitCo wrote up (https://blog.lopp.net/the-challenges-of-building-ethereum-infrastructure/) a detailed blog post outlining some challenges they ran into with their own low-level smart contract infrastructure and surrounding applications. Gladly, almost two years later, at least the library ecosystem seems to have improved considerably.
- Infura (https://blog.infura.io/building-better-ethereum-infrastructure-48e76c94724b) published some insights into how their cloud infrastructure works, and the additional services they needed to implemented to support their feature set efficiently.
- EtherVM (https://ethervm.io/) is an EVM decompiler which is able to turn raw EVM bytecode into a readable low-level pseudocode format without access to the original source code. The web site is also a great resource for learning about EVM internals.