Privacy Issues and Concerns on Ethereum Network Layer

2018-09-29T14:30:00-04:00

Contents

Introduction

Ethereum's network layer, or more precisely devp2p (or libp2p) while providing **encryption** and **authentication**, does not necessarily give user **anonymity** or **privacy**. In this post I will point out some of the privacy issues, and potential attacks to de-anonymize Ethereum account owners (i.e. associating an Ethereum account address with the IP address of its owner).

> For simplicity we do not consider anonymity networks during attack like *Tor* or *I2P*, although they will be discussed in the Mitigation section.

Background

Feel free to skip this section if you already know how ultralight clients (like *metamask*) work.

Layers and stacks

While it is not a formal definition, the Ethereum stack can be vaguely divided into the following layers:

1. Smart Contract Layer (layer 2)

- Smart contracts are deployed onto the blockcahin and its code becomes immutable after deployment
- Transactions can be sent to Smart Contracts, and the recipient Smart Contract code determines what actions should be taken
- For example, layer 2 scaling solutions like *plasma* require interaction with this layer, without changing the consensus layer
- Examples include ERC20 token contracts, and multisig wallet contracts

2. Consensus Layer (layer 1)

- This layer determines what transactions and blocks are valid and what are not
- Changes in this layer would typically require *hard forks*

3. P2P Network Layer (layer 0?)

- Participants of Ethereum (e.g. full nodes running *geth* or *parity*) connect to each other through the peer-to-peer network
- Transactions and blocks are broadcasted in the peer-to-peer network

4. Protocol Implementation Application Layer

- Or "node" applications, these applications implement the Ethereum protocol
- Nodes running these applications connect to each other through the P2P network
- These applications typically run a JSON-RPC server, allowing wallet applications to send transactions or read chain state
- Examples include *geth*, *parity*, *ganache-cli (formerly testrpc)*

5. RPC Network Layer

- A JSON-RPC client would connect to a JSON-RPC server through this layer
- Typically HTTP or HTTPS, but also includes IPC
- Wallet applications connect to a protocol implementation application through this layer
- For example, `web3.js` is a library for wallet application to JSON-RPC call a protocol implementation application.

6. Wallet Application Layer

- Applications on this layer typically RPC calls the protocol implementation applications (e.g. *geth*) to accomplish certain front end tasks.
- Those applications can potentially connect to a *protocol implementation application* through HTTP or HTTPS over the Internet.
- Examples include wallet applications like *mist* or *metamask*, where transactions are signed in a wallet application and the signed transactions are sent to a node running *geth* via HTTP or HTTPS.

  The interest of this post is both the *P2P Network Layer* and the *RPC Network Layer*.

Types of clients

There are different types of clients that can sign or send transactions, or connect to peers, or receive blocks, or verify transactions.

Full node client

We say a user Alice is running a **full node client** if Alice is:

- Running a "node", i.e. a protocol implementation application like *geth* on her machine, and
- The said protocol implementation application is running in "full" mode, and
  - In the case of *geth* this means running `geth --syncmode=full`, which is the default
- Running a wallet application like *mist* or *metamask* on her machine
  - The wallet application connects to her node through HTTP localhost
  - Alice can also use her *geth* as wallet

A full node downloads the entire blockchain data. It gets the block headers, the block bodies, and validates every element from genesis block.

> In this case the *RPC Network Layer* is not particularly interesting as it is just local socket.

Light node client

(Also known as SPV client)

We say a user Alice is running a **light node client** if Alice is:

- Running a "node", i.e. a protocol implementation application like *geth* on her machine, and
- The said protocol implementation application is running in "light" mode, and
  - In the case of *geth* this means running `geth --syncmode=light`
- Running a wallet application like *mist* or *metamask* on her machine
  - The wallet application connects to her node through HTTP localhost
  - Alice can also use her *geth* as wallet

A light node does not download the entire blockchain data. It gets only the current state. To verify elements, it needs to ask to full (archive) nodes for the corresponding tree leaves.

> Similar to full node, *RPC Network Layer* is also not interesting to us here because it is just local socket.

Ultralight client

(Also known as zero client)

We say a user Alice is running an **ultralight client** if Alice is:

- NOT running any protocol implementation application like *geth* on her machine, and
- Running a wallet application like *mist* on her machine
  - The wallet application JSON-RPC connects to another full node on the Internet over HTTP or HTTPS

> **Fun fact**: *metamask* is an ultralight client by default, which JSON-RPCs `https://mainnet.infura.io/metamask`, a full node running *geth*, operated by Infura.

One important thing to note is that an ultralight client learns everything from a node (full or light), and lacks the capability to verify transactions or blocks.

Privacy issues

In this section we discuss a few scenarios where account holders can be de-anonymized, or their privacy can be violated. Generally, they fall into these two categories:

- Privacy issues during **transaction broadcast**
  - On RPC network layer, and P2P network layer
- Privacy issues during **chainstate retrieval**

– On RPC network layer, and P2P network layer

Broadcasting a transaction as an ultralight client

In this scenario Alice who is running an ultralight client, is sending a transaction which moves some funds from Alice's account to Bob's.

A few things happen in order:

- Alice's ultralight client reads Alice's private key from Alice's computer
- Alice's client generates a transaction, that says "move funds from Alice's account to Bob's"
- Alice's client uses Alice's private key to sign the transaction
- Alice's client JSON-RPC calls a full node over HTTP, in this case a node controlled by Eve, to do `eth_sendRawTransaction`, passing along the signed transaction
- Eve's full node broadcasts the transaction to peers over P2P network
- Eventually miner picks up the transaction, putting it into a block
- Miner broadcasts the found block over p2p
- Eve's full node receives the found block over p2p
- Alice's client RPC calls Eve's node `eth_getTransactionReceipt` to confirm that her transaction has been mined

In this simple case, it is pretty obvious that Eve can learn Alice's **Ethereum account address**, her **public key**, transaction **recipient's account address**, **the value** of the transaction, along with her **IP address** from the RPC network layer. Alice has **zero privacy or anonymity** as Eve can see all the transactions she sends. It can be worse if Bob also uses an ultralight client and connects to Eve's node!

Real world implication

If you use *Metamask* with default RPC settings, *Infura* can dox you whenever you send a transaction. And they learn all the details about the transaction.

Keep this in mind before conducting your business!

Retrieving chainstate as an ultralight client

Observation: An ultralight client frequently calls `eth_getBalance` with their account address on a full node to keep the account balance up to date in the wallet application UI.

Over time, Eve can observe that, the wallet at Alice's IP address frequently requests the balance of a set of addresses, via `eth_getBalance` RPC call. With more sophisticated temporal analysis, Eve can learn all the accounts Alice owns in her wallet software.

Consider a more interesting scenario involving smart contracts:

- Bob runs a lottery service powered by Smart Contracts
- Bob's web3 site has no trackers. On the front page of his site it simply shows the current jackpot
- Alice simply browses Bob's lottery site with *Metamask* enabled, without playing the lottery
- Alice's browser uses `web3.js` to get jackpot info (here Metamask injects a global web3 object into browser)
- Alice's *Metamask* issues a `eth_call` RPC on Eve's node to query jackpot info
- Eve learns everything about the query along with personal identifiable information about Alice.

In this case Eve can learn exactly how many of her users have been on this site, along with their account addresses and IP addresses, without needing any analytic data from Bob. Because in order to read the jackpot data from Bob's contract, Alice's browser runs the JavaScript Bob has written, which essentially does `eth_call`, to query the jackpot info on Bob's contract. Eve in this case learns the **origin of the RPC call** (Alice's IP, account address), the **contract address**, **the function being eth_call'ed**, and the **state of the contract** during the call.

How Eve might use those data is outside the discussion of this post, but Alice is essentially making her web3 browser history public to Eve!

Real world implication

If you use *Metamask* with default RPC settings, even if you do not send any transactions through the ultralight client, *Infura* can still learn addresses of all the accounts you own over time. They will even learn what smart contracts you are interested in!

This is like making your browser history public!

Broadcasting a transaction as a full or light node client

Observation: The first node to broadcast a transaction is typically the origin of the transaction.

In this case Alice finally runs her own node. But her privacy is still in danger of being violated by the peers her node connects to on the P2P network layer. However this isn't as bad as Alice running ultralight client and using Eve's full node to broadcast.

Alice's full or light node is connected to some peers. She signs a transaction in her *Metamask*, then her *geth* broadcasts it to all her peers. Her peers upon

receiving this transaction, put it in their txpool, and propagate the transaction to other peers in the P2P network.

In this case, Alice's peers will learn that the transaction came from Alice's node. Without more data, they cannot tell if Alice's node is the origin of the transaction, as she might simply be relaying the transaction from someone else. However, a more powerful passive attacker who controls a lot of nodes can eventually figure out that Alice's node is the origin, with some degree of certainty. Traffic analysis can also be used here to dox Alice.

> **Tip**: In *geth* console you can run `eth.net.peerCount` and `admin.peers` to check your peers. You will see their node application version, OS information, and their IP address. They can see yours too!

Real world implication

geth (or its p2p layer devp2p) does not come with anonymity network implementation. Your peers can potentially learn all the transactions you are signing and broadcasting, and associate those with your IP address!

If you want to conceal your wealth (i.e. not letting anyone learn that it is you who control a particularly wealthy account), it's probably not a good idea to run *geth* without an anonymity network like I2P!

Retrieving chainstate as a light node client

This only applies to light node, i.e. node running in light sync mode. Full nodes download the entire database, therefore do not have this problem.

Suppose Alice is running a light node. Consider the same scenario where she visits Bob's lottery site.

- Alice's *Metamask* issues an `eth_call` on her own node
- Alice's node does not have enough information to complete the call, since it is running in light mode.
- Alice's node requests the state she's retrieving along with proofs for SPV (*Simple Payment Verification*) from peers. For simplicity we assume Alice's peers are all full nodes.
- The peer Alice sends the request to learns the particular state Alice is looking for, along with her peer ID, IP address, etc.

This is a similar case to "*Retrieving chainstate as an ultralight client*", except here Alice asks her peers *vaguely* what she wants, and the peers are discovered and chosen through a peer discovery process, which makes targeted attacks more difficult. However it is not as difficult as you think, as we will discuss in the next section.

Active attacks

In this section I want to briefly discuss how those issues can be exploited if an active attacker is on the scene. Our active attacker, Mallory, wants to see what Alice is up to.

Peer discovery poisoning

When a node joins the P2P network, there are a few so-called **bootstrap nodes** that are hardcoded into the protocol implementation application such as *geth*. Those bootstrap nodes help peers find each other. When Alice joins the network and attempts to connect to the bootstrap nodes, an active attacker Mallory (e.g. maybe Alice's ISP) can block the connection, so she does not get to randomly pick peers from the bootstrap node. When Mallory's nodes connect to Alice as her peers, Alice's node happily accepts them. Of course, devp2p is designed such that even without bootstrap nodes, the peers would still be able to find each other eventually, but since Mallory's nodes are connected to Alice's first, they are less likely to be dropped by Alice (unless they are broadcasting invalid transactions or blocks).

If Mallory is more powerful, they could simply coerce the bootstrap nodes to feed Alice a modified version of the peer list containing only Mallory's nodes.

Once Mallory has enough peers on Alice, Mallory can see all the transactions she's broadcasting or relaying, as well as all the chainstate her node is requesting.

Traffic analysis

An active attacker can analyze Alice's **RPC network traffic**, or **P2P network traffic**, or **both combined**.

If Alice runs a full node (say, on her desktop) but her wallet is running on a different machine (say, on her phone), and her wallet RPCs over HTTPS over the Internet. Mallory, despite not being able to see the request, can see some matadata about the RPC packets, like the size, destination IP, etc. Without seeing Alice's full node's P2P traffic, Mallory can already potentially find out what Alice is doing over time using traffic analysis techniques.

Same thing applies to her P2P traffic. If Mallory can see all the traffic going into and coming out of her node, even though the packets are encrypted to peers, Mallory can still learn a lot over time using traffic analysis techniques.

I will leave out the details as traffic analysis is a pretty sophisticated subject. Of course, this type of attack is harder to pull off without lots of resources.

Mitigations

We can already see that running an ultralight client is pretty bad for privacy (security issue aside). Running a light node or a full node doe not guarantee us privacy either. In this section some mitigation techniques will be discussed, with their pros and cons.

Anonymous overlay networks

In practice, *Monero* node uses Kovri, an implementation of I2P. When a transaction is broadcasted, it can be difficult for a passive attacker to find out the origin of the transaction without a global view of the network.

Whenever some information is retrieved, a passive attacker will not be able to pinpoint the exact peer who requested it. Of course, the attack would still learn the content of the data being retrieved, along with timestamp and other metadata. This makes user susceptible to traffic analysis.

Of course, no anonymity network can be perfectly anonymous.

> *The continued goal of I2P is to make attacks more and more difficult to mount. Its anonymity will get stronger as the size of the network increases and with ongoing academic review.* - I2P The Invisible Internet Project

Recommendation

Always be on I2P before joining P2P network. You can try getting *Kovri* on the machine running *geth*.

(I don't think you can use geth over tor, since geth uses some UDP ports and tor is TCP only.)

Retrieving entire database

To retrieve data from a database without revealing which item is retrieved is a private information retrieval (PIR) problem. One trivial, but very inefficient way to achieve PIR is for the server to send an entire copy of the database to the user. This gives user *information theoretic privacy*, without the assumption of non-colluding servers.

A full node does exactly this. A full nodes downloads every transaction and block ever existed from the genesis block. Peers simply do not have the information to learn which transaction or block you are actually looking for.

The drawback of course, is that the full node would have to download and store lots of data, which might be infeasible on devices like smart phones.

Recommendation

Always run your own `geth`. Always run a full node. Use `--syncmode=full` for the best privacy, but `--syncmode=fast` is also OK. Connect your wallet to your node using local socket, so you are not susceptible to RPC traffic analysis.

Bloom filters

> Currently Ethereum block headers include *logs bloom*, which is a *bloom filter* for all events emitted in transactions in the block. This is useful for nodes to quickly find interesting transactions that emit logs. In EVM logs are emitted using `LOG0-LOG4` opcode. But this is a different use case of bloom filter we discuss in this section.

Bloom filter is a probabilistic data structure to test if an item is "probably in a set". It has false positives but never false negatives.

Bitcoin SPV clients can create a bloom filter, put all the transactions it wants to query in the filter, then sends the filter to a full node. Then the full node tests transactions against the filter, if it is "probably in the set", it returns the transaction data plus proof to the SPV client. To preserve some degree of privacy, the said SPV node can set up the bloom filter to have high false positive rate, so the full node does not know which transaction the SPV client is actually interested in.

Although not implemented, light nodes on Ethereum can potentially also use this bloom filter technique to retrieve transaction data or state data from full nodes to preserve privacy. But bloom filter is only privacy preserving if client randomly samples nodes from peer list to send filters to each time, and the selected full nodes do not collude. Over time a full node would be able to see patterns from the bloom filters sent by a particular client, e.g. matching transaction graph or finding common addresses among sets. **The privacy-preserving property of bloom filter is weak** but not entirely useless.

Future research

Blockchain PIR solution where a node could achieve privacy without having to download the entire blockchain database would be a potential topic of research.

Appendix

IPFS privacy

IPFS is a P2P hypermedia protocol that aims to make the web faster, safer, and more open. However IPFS suffers from the same privacy issues.

- When you requests some data on IPFS, your gateway knows exactly what you are requesting, and the peers the gateway node connects to also learn the request.
  - So IPFS is not a good solution to download pirate movies or games
- The first node to serve the file is typically the origin of the file. This can be used to pinpoint the exact peer that is hosting the file.
  - So no, IPFS alone is not a very good solution for whistleblowers to upload classified documents

In some ways privacy on IPFS is worse than today's web. At least today only you and the web server you are connecting to know what you are looking at on the site. (There's a bunch of other privacy problems with IPFS, but maybe in another blog post?)

References

1. https://github.com/libp2p/specs
2. https://medium.com/l4-media/making-sense-of-ethereums-layer-2-scaling-solutions-state-channels-plasma-and-truebit-22cb40dcc2f4
3. https://ethereum.stackexchange.com/questions/11297/what-is-geths-light-sync-and-why-is-it-so-fast
4. https://github.com/ethereum/wiki/wiki/JSON-RPC
5. https://geti2p.net
6. https://en.wikipedia.org/wiki/Private_information_retrieval
7. https://petsymposium.org/2015/papers/14_Borisov.pdf
8. https://github.com/ethereum/wiki/wiki/Light-client-protocol