What's that touchscreen in my room?

2024-01-20

> *Discussion on HackerNews and Lobsters.*

Roughly a year ago I moved into my new apartment. One of the reasons I picked this apartment was age of the building. The construction was finished in 2015, which ensured pretty good thermal isolation for winters as well as small nice things like Ethernet ports in each room. However, there was one part of my apartment that was too new and too smart for me. This thing:

It is obviously a touchscreen of some sort, but there was zero indication as to what it controls. The landlord had no idea what this is. There are no buttons or labels on the thing, just a tiny yellow light to let you know it has the power.

I had a million questions, but I was too busy with the move and kinda of forgot about it until about a week ago. I was looking through a huge binder of various appliance manuals for my apartment when this thing slided out:

Wait a second, that's my touchscreen! Turns out it is a part of an energy monitoring system, which tells you current energy usage and has the ability to display historical data. That actually sounds pretty neat - I would be interested to see energy usage patterns of my house.

Brochure also mentioned a second part of the so-called "energy manager", which was directly plugged into an electricity meter to get usage information. I went to inspect communal cupboard housing electricity meters - and sure enough, there it was, just standing in the corner. I never even

noticed these boxes before, but they at least have the same branding as the brochure - some company creatively named “NETTHINGS”.

Pretty straightforward so far. We have two devices, one “server” gathering the data and another one “client” reading the data. Now the distance between them is actually very short - just a few meters and maybe 2-3 walls, totally reasonable setup for a cable connection. It was at that moment when I noticed a weird sticker in the corner of the brochure. There were two strings printed with labels “SSID” and “Pwd”. I froze in horror. They wouldn’t dare. It is literally 3 meter distance. These are embedded devices, they do not need this complexity…

And of course the two devices communicate using WiFi. Now that was unusual for me, since I am not an embedded developer. But a friend of mine, who worked on smart home features for one voice assistant told me that this is actually a pretty common thing to do in IoT space. C in IoT stands for “cost-effective” I guess.

Moving on, I needed to somehow turn on the weird touchscreen. Upon closer inspection, I noticed a small hole on the side, which looked a lot like something you insert a pin into to reset the device:

I held a hidden button with a wire for a few seconds and was greeted by…an Android bootup logo! Yes, this turned out to be in fact an Android tablet, and a pretty old one at that. It has Google Talk, Flash and all kinds of other interesting stuff pre-installed:

From what I can tell, this is an Android 5, but I am not exactly sure. One of the apps stands out with a familiar name: “NetThings”. Launching it leads us to the screen where we select a WiFi network. Unfortunately, the one that is mentioned on the brochure, was not on the list. I double-checked the list, tried to refresh it, looked for WiFi networks using my other devices, tried to directly connect to the WiFi using the credentials, but no luck.

When I came back to the meter room I noticed the obvious: boxes for all other apartments had the light on, but mine was off for some reason:

I live in the UK and if anything, this is a country of electrical fuses. The more fuses the better, everything is fused, even some plugs. Fuses for certain appliances come in separate boxes with a small drawer that can slide out, allowing to replace the fuse. It looks like this:

As you can see, my fuse box did not have a fuse inside for some reason. No fuse - no electrical connection, no power for the energy manager and no WiFi hotspot. Thanks to the fuse box design, it is easy to replace one, but I did not know what kind of amperage should be allowed in the circuit. Luckily, I had a few working energy managers in the same cupboard, so I opened their boxes (cutting the power to them for a few seconds) to see what kind of fuses they use. Turns out, I need a 3A fuse, so I ordered one from Amazon and installed it the next day.

To be honest, the whole thing was a bit scary, since I was very close to the mains. After installation, I checked the temperature of the fuse multiple times during the day to get at least some indication that things are not

going to get worse. It worked fine for a more than a week now, but I still do not recommend experiments like this to anyone.

After installing the replacement fuse, energy manager started blinking with green LEDs and the promised WiFi network appeared on all devices. Once I selected the network on the Android tablet, it changed to the following screen:

When tapped, it changes to this menu, inviting the user to select what kind of resource they want to monitor. Nothing here actually works except for “Mains Electricity” because that’s the only meter the energy manager is hooked up to.

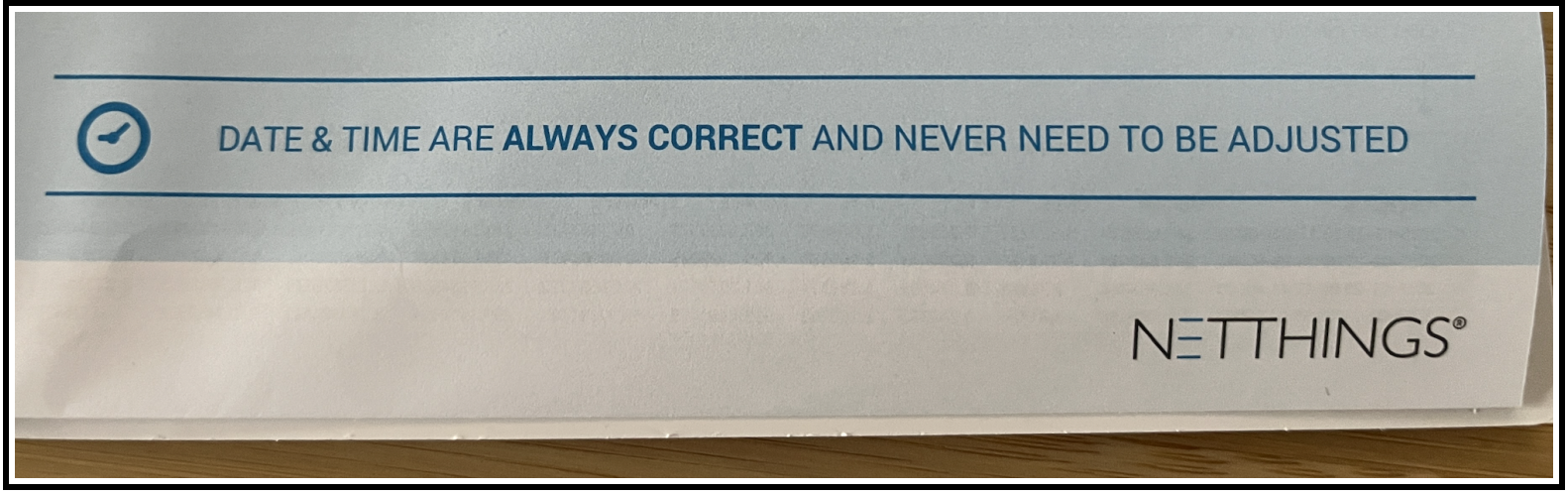
“Mains Electricity” leads us to the most dissapointing screen in the history of UX design:

Ugh, where to start. What's up with this color indicator on the right? What does the vertical position mean? If it is green, does it mean that I am using a small amount of electricity or just a normal one? What exactly is small? If it gets all the way to the top, is it compared to my historical maximum usage? Over what period of time?

Out of 5 numbers displayed to the left of the indicator, only one is actually true - number of kW consumed. All other numbers depend on the energy provider and definitely changed since the time this monitor was installed.

I saved best for last. The amount of money you pay as well as estimate of CO2 per kW **is not configurable**. According to the brochure, it was configurable during the initial installation, but it has no information on how to reset the system back into configurable state.

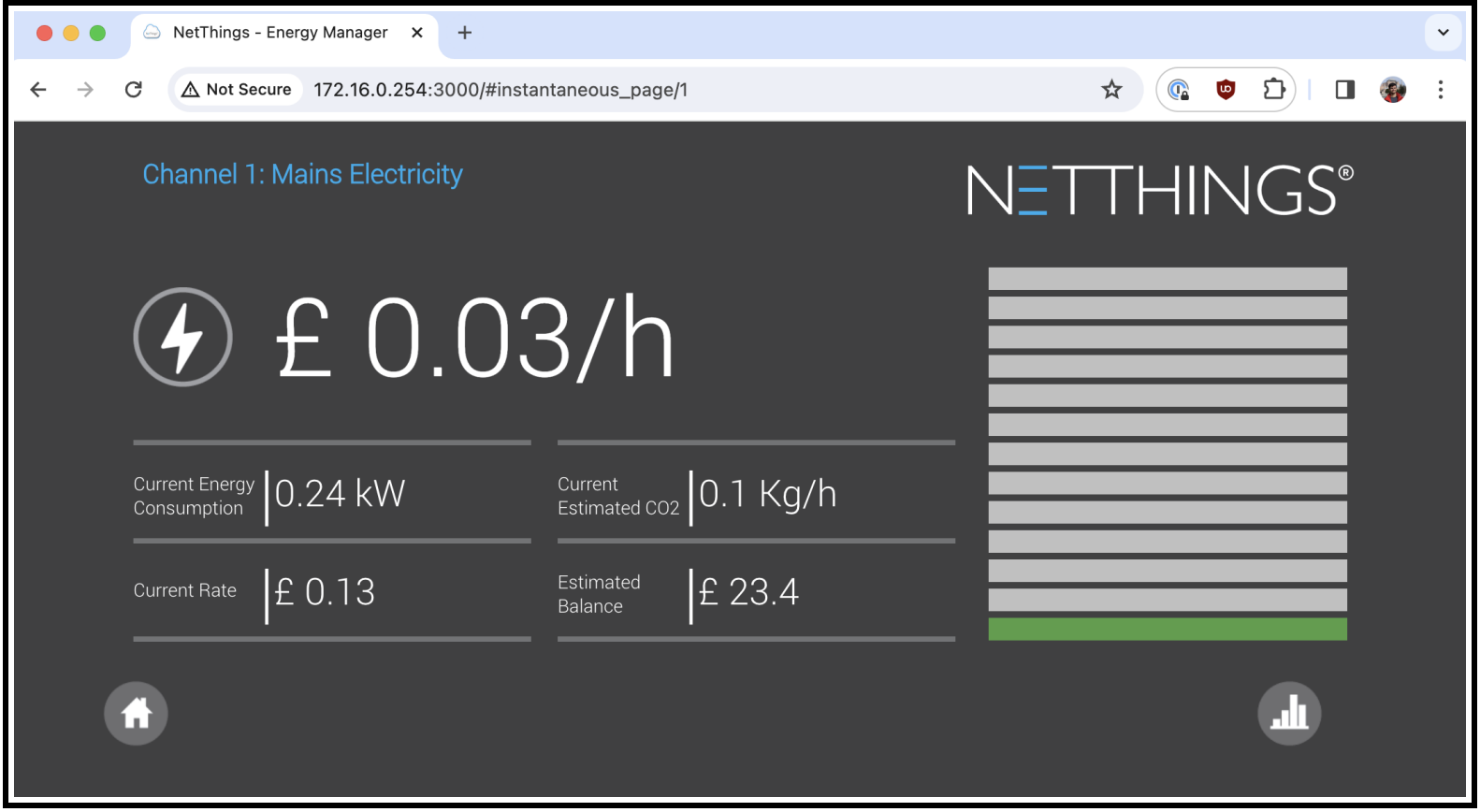
Finally, brochure says the following:



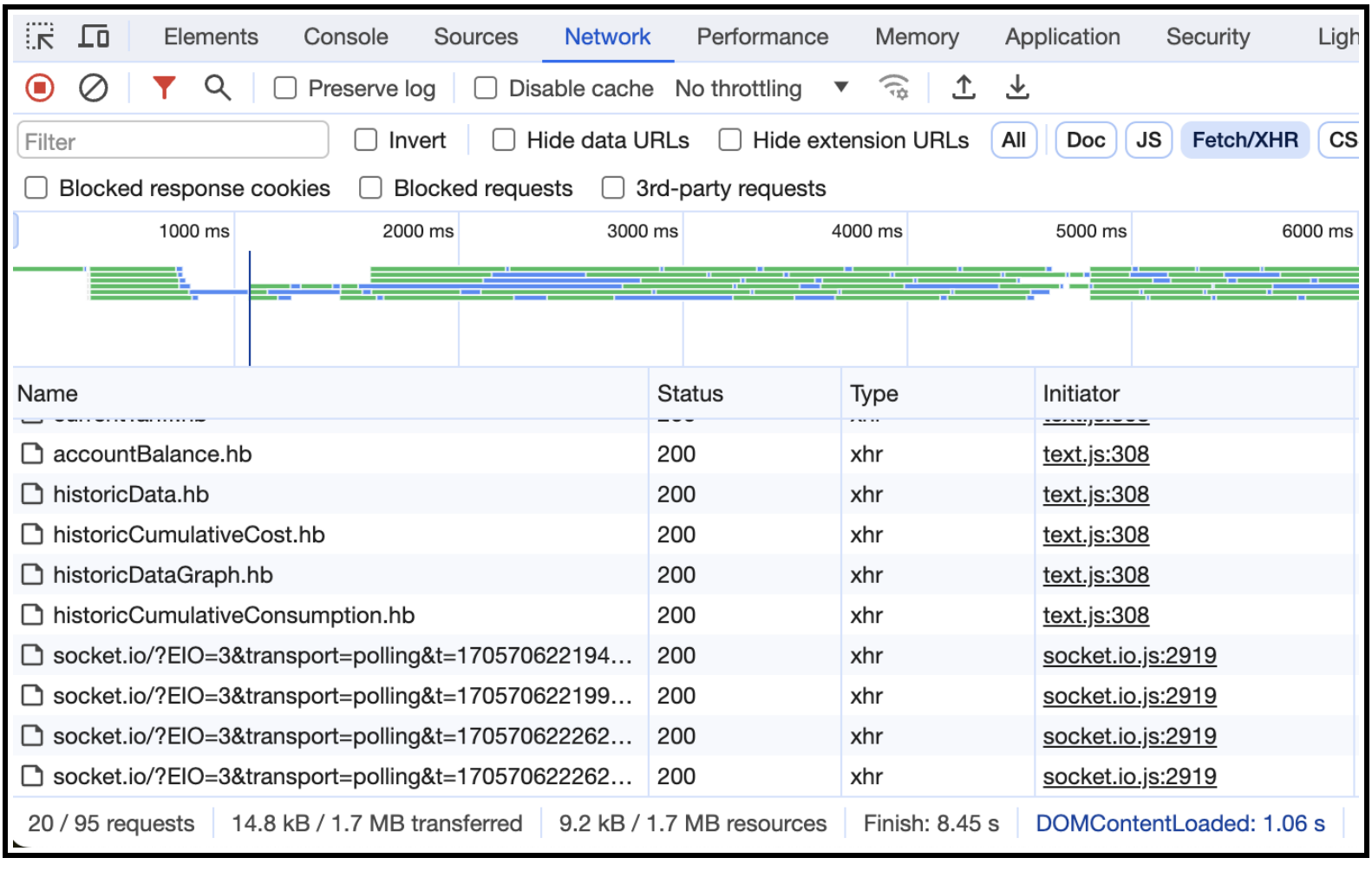
I have no idea why they decided to include this. Of course the clock on Android tablet shifted by almost 15 minutes since 2015, when it was supposedly installed.

The whole thing was quite dissapointing. However, I do have a few Raspberry Pico microcontrollers lying around at home. If I could connect to the WiFi network of the energy manager directly and get the data from the server, I could just extract kW consumption from the API, multiply it by a correct rate and then display it on some Grafana instance.

The main problem was that I do not know the IP of the server. I was just about ready to launch a full IP scan from laptop when I noticed that one of the use cases brochure advertised is checking the energy usage from the PC. The IP and port were conveniently provided together with the instructions. Opening it in the browser displays a familiar screen:



Turns out, interface on the Android tablet is just a webview. This makes our job that easier, since we can just go to the web inspector and see all the API calls. Looking at the URLs we see…



…Socket.IO! Wow, I honestly did not expect that. Client literally needs to receive 5 numbers from the server, Socket.IO seems to be a complete overkill for this usecase. The client code also looks very complicated for what it does. There are at least 6 RequireJS modules, all loading dynamically through different requests of course. There is Handlebars, Backbone.js, Underscore.js… I feel like I am in high school again. These are all technologies I was very interested when I just started web development.

But wait a second, Socket.IO would mean that an embedded device in my meters cupboard is running JavaScript? This must be weirdest Edge Computing Platform I have seen in my life. I want to deploy *something* there!

Totally forgetting the idea with Raspberry Pico fetching the data from the server, I put on my hacker hat and started poking around. IoT devices have a terrible reputation from the security perspective, so I expected it to be an easy run for a couple of hours tops. Boy was I wrong.

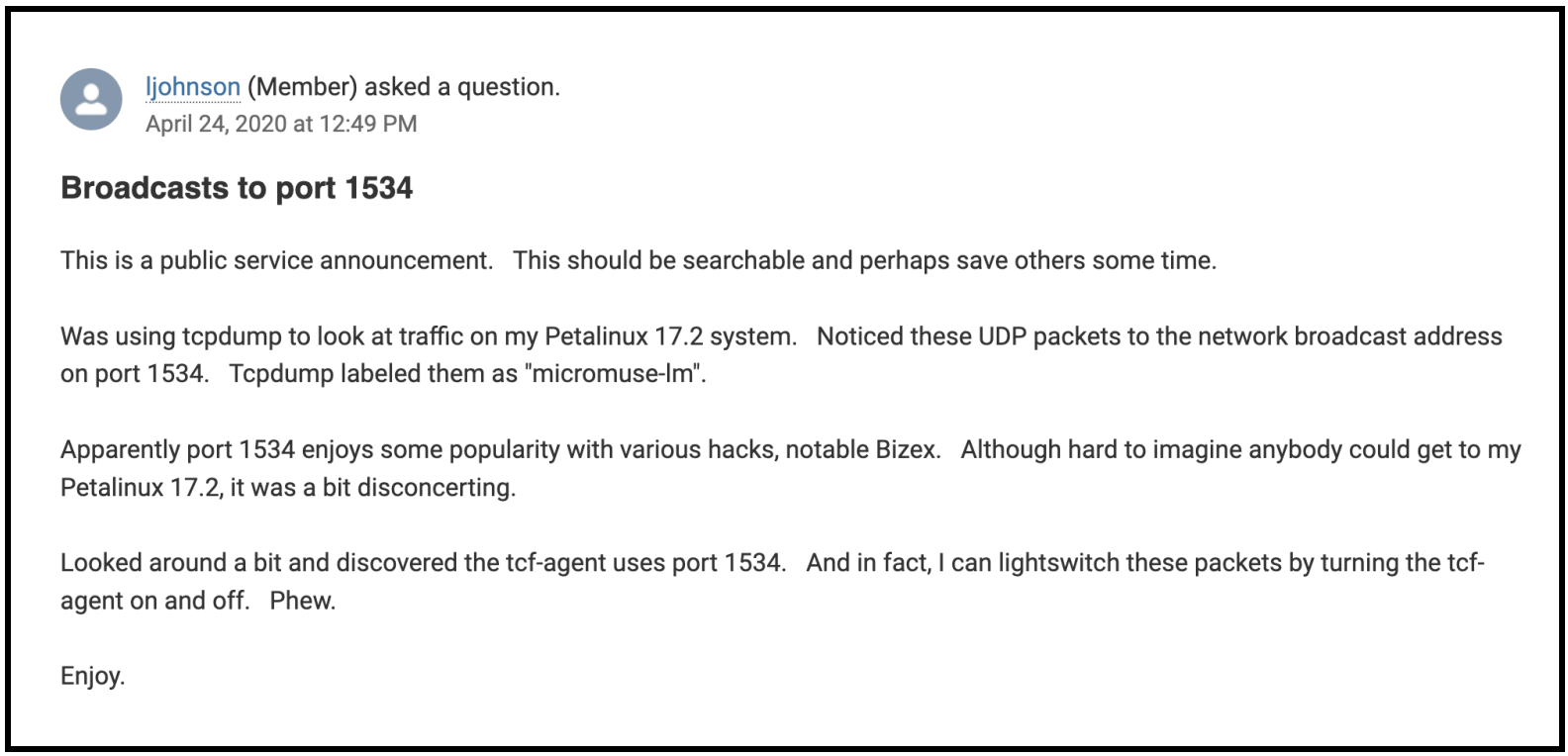
Direct SSH using `ssh root@172.16.0.254` immediately fails with "Connection refused" error. That could mean a number of things, let's see which ports are available:

```
$ sudo nmap -p- -sV -O 172.16.0.254
... truncated output ...
Nmap scan report for 172.16.0.254
Host is up (0.011s latency).
Not shown: 65530 closed tcp ports (reset)
PORT      STATE SERVICE       VERSION
53/tcp    open  domain        dnsmasq 2.63rc6
80/tcp    open  http          Node.js (Express middleware)
1534/tcp  open  micromuse-lm?
3000/tcp  open  http          Node.js (Express middleware)
41142/tcp open  ssh           OpenSSH 6.2 (protocol 2.0)
```

Now this is interesting. As expected, we see a Node.js server running. We also have `dnsmasq`, which is a DHCP server (makes sense, since the device is a WiFi access point) and a hidden SSH server on port 41142.

SSH connection is no longer refused, but the root turned out to be password-protected. None of the simple username/password combinations like `admin/admin` or `root/root` worked, so we are essentially back to square one.

However, nmap detected an unrecognized service on port 1534 called micromuse-lm. The first Google result is the following forum post:



ljohnson (Member) asked a question.
April 24, 2020 at 12:49 PM

Broadcasts to port 1534

This is a public service announcement. This should be searchable and perhaps save others some time.

Was using tcpdump to look at traffic on my Petalinux 17.2 system. Noticed these UDP packets to the network broadcast address on port 1534. Tcpdump labeled them as "micromuse-lm".

Apparently port 1534 enjoys some popularity with various hacks, notable Bizex. Although hard to imagine anybody could get to my Petalinux 17.2, it was a bit disconcerting.

Looked around a bit and discovered the tcf-agent uses port 1534. And in fact, I can lightswitch these packets by turning the tcf-agent on and off. Phew.

Enjoy.

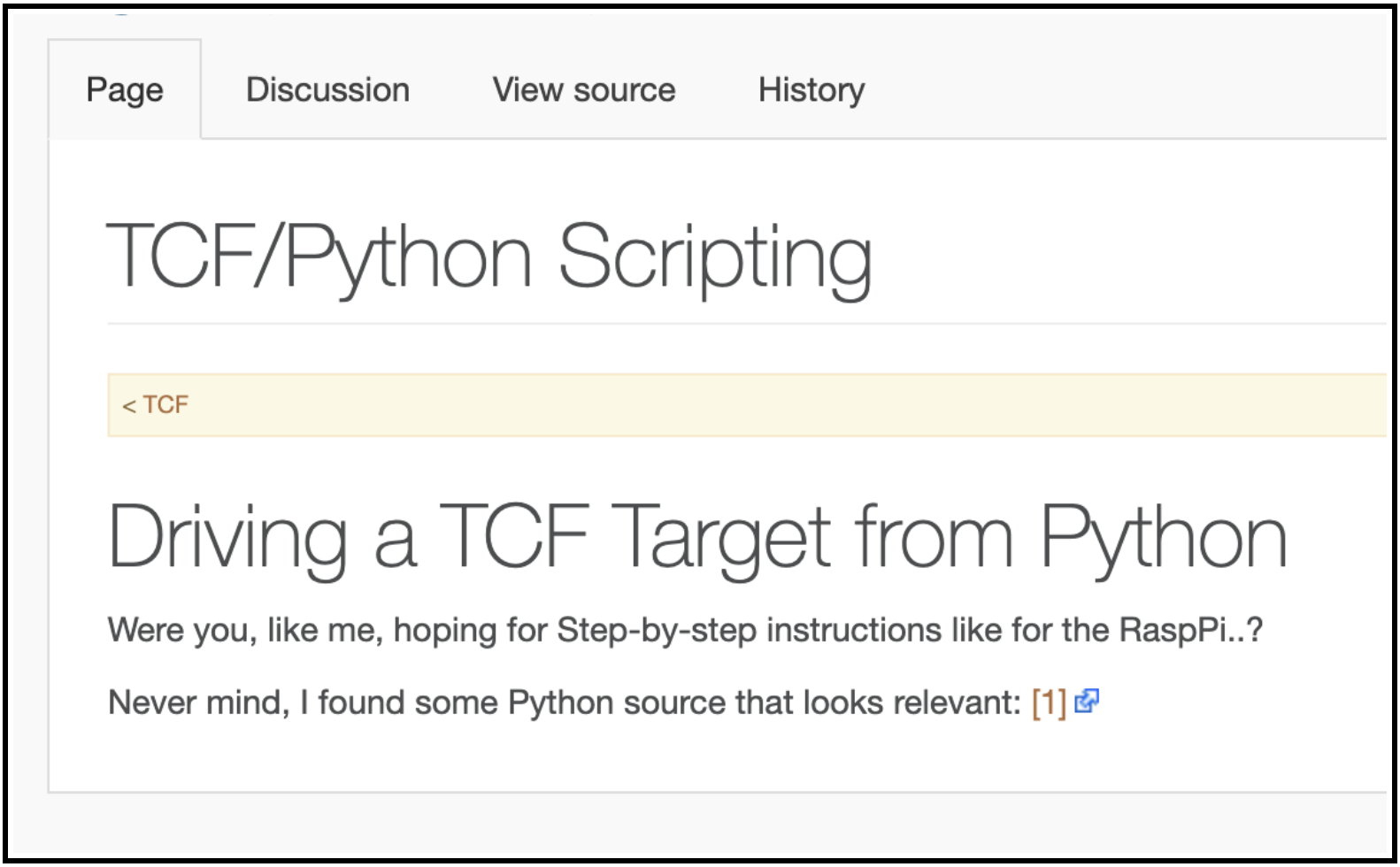
I do not know who you are @ljohnson, but may your life be happy and prosperous. This post does not give a lot of information, but it provides one with the correct keywords to continue the search. The main phrase here is tcf-agent. The concrete description of what is going on here is spread atom-thin across several websites, all of which expect you to know the terminology. Each of these websites provides you with a tiny piece of the puzzle and you are expected to combine it together on your own. So after a few hours and lots of cursing, here is what I know about tcf-agent:

TCF stands for "Target Communications Framework". It is a text protocol, which allows to read the filesystem, start new processes, send signals to processes and a lot more with the target system. tcf-agent is the server implementing this protocol or, in another words, probably the second biggest security vulnerability after passwordless root SSH. I do not understand why they went into all this trouble with SSH passwords, but kept tcf-agent running.

TCF seems to be closedly tied to Eclipse ecosystem. The Getting Started guide suggests several plugins for Eclipse as the main way to interact with tcf-agent. I tried installing these plugins on a new version of Eclipse and it is absolutely impossible. There are dependency issues everywhere and when you actually try to install the missing dependencies, Eclipse does not let you

because they conflict with some other dependencies. It's a mess, which is exactly how I imagined this interaction to go.

Conveniently, TCF project has a Python SDK. As with everything during this research, I needed to go through 3 links on different websites to actually find one. First, we are met with this lovely page:



Which has a link that leads to here, which has a very small text pointing out that the repo moved to this Gitlab repository. Phew. The repo even has a few fresh commits, which seems a bit suprprising to me, as TCF in general gives off vibes of an abandoned project.

Nevertheless, the repository contains a pretty modern Python 3 (!) SDK, even with some inline documentation. It is not perfect, some docs are outdated, some methods are very weird, but you can pretty easily figure out what's going on from the code. Protocol specification here and here are a huge help in this process.

In a nutshell, a `tcf-agent` provides a bunch of services that expose various parts of the system. For example, there is a `FileSystem` service for all interactions with the filesystem, `Processes` service for

starting/stopping/debugging processes, etc. Here is an example of getting current user information:

```
import tcf
from tcf.util.sync import CommandControl

tcf.protocol.startEventQueue()
cmd = CommandControl(tcf.connect('TCP:172.16.0.254:1534'))
error, user = cmd.FileSystem.user()
print(user)
```

And that's how we learned that `tcf-agent`, in fact, runs under the `root` user. Again, why bother with SSH passwords if you leave a debug server with root access - this I will never understand.

`FileSystem` and `Processes` services have other functions, roughly corresponding to syscalls. You can pretty easily replicate alternatives to common commands like `ls`, `cat`, `ps` and the like using this API. Now I say "pretty easily", but in reality that was 4 hours of guessing the protocol format, trying to the find the documentation, fixing bugs in the SDK, all on an extremely unstable WiFi connection from an embedded device. Fun times. You can find the results in this Github repo.

Now that we have basic instruments, let's get to hacking! My first attempt was to crack the root password, as I still believed it was something trivial. I used John the Ripper password cracker in the following way

```
# `cat.py` fetches file contents from the energy manager
# using `FileSystem` TCF service.
$ ./cat.py /etc/passwd > passwd.txt
$ ./cat.py /etc/shadow > shadow.txt
$ unshadow passwd.txt shadow.txt > passwords.txt
$ john passwords.txt
```

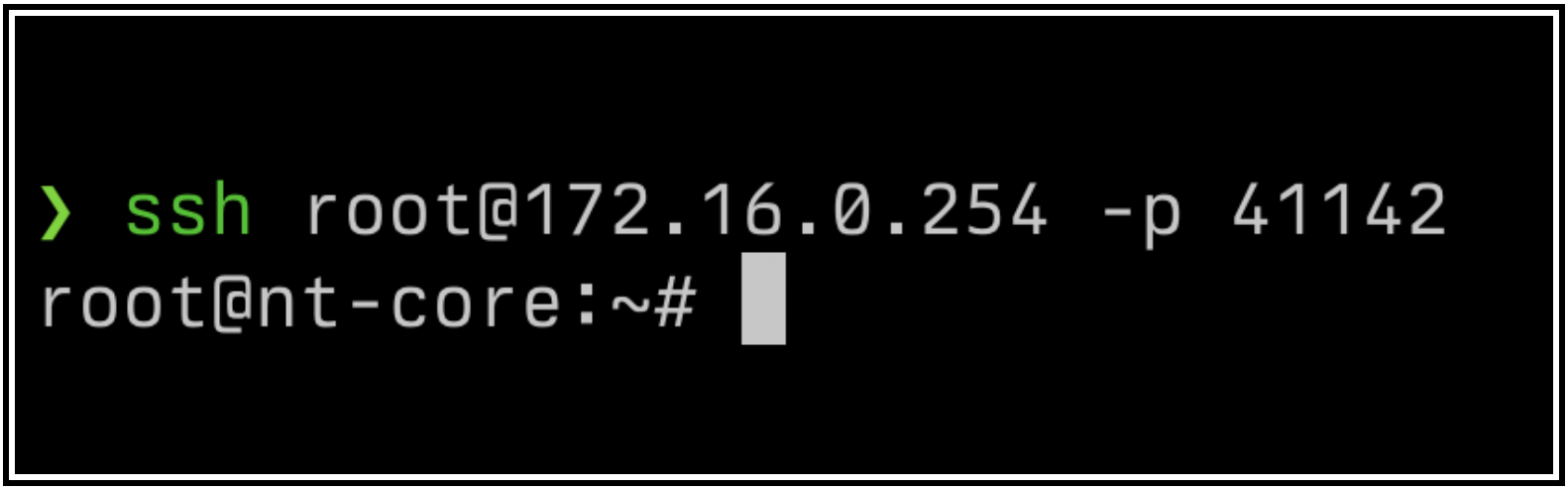
I left it running for roughly 7 hours, but it did not find a match. John reported that it will finish its brute-force in the year 2035, so I decided to try out a different approach.

After a bit of Googling, I found out that one can simply make root's password empty by modifying `/etc/shadow`. Having done just that, I powercycled the

device by removing the fuse I installed previously and putting it back after some time. Unfortunately, SSH still rejected my login attempts.

More out of desperation than anything else, I decided to look at sshd config of the host and finally found the offending line. `sshd_config` had `PermitRootLogin no` line included, which is a very sensible security measure as long as you are not providing a full disk access to anyone on the network.

I replaced the line with `PermitRootLogin yes` and finally, I saw the output I was struggling for:



We are in! God, that was quite a journey, wasn't it? Let's look around!

```
root@nt-core:~# uname -srm
Linux 3.10.28 armv5tejl
```

We can see that we are running Linux 3.10, which is actually quite a recent release (middle of 2013), considering this device was developed and installed roughly in 2014-2015. Unsurprisingly for an embedded device, it is powered by an ARM chip:

```
root@nt-core:~# cat /proc/cpuinfo
processor       : 0
model name      : ARM926EJ-S rev 5 (v5l)
BogoMIPS        : 226.09
Features        : swp half fastmult edsp java
CPU implementer : 0x41
CPU architecture: 5TEJ
CPU variant     : 0x0
CPU part        : 0x926
```

```
CPU revision    : 5

Hardware    : Freescale MXS (Device Tree)
Revision    : 0000
Serial      : 0000000000000000
```

We have an ARM9 family CPU. Wikipedia says that it was released in 2001, with a list of notable mentions including being a coprocessor for Nintendo Wii.

The fact I find even more surprising is that this processor supports executing Java bytecode directly. Yep, you read that right, the `java` in the list of CPU features actually means **that** Java. The ARM extension for this feature was called Jazelle. This seemed to be some kind of a trend around 2000s, since this is not the first time I encounter such feature.

```
root@nt-core:~# cat /proc/meminfo
MemTotal:         118172 kB
...truncated...
```

Lastly, we have 118MB of RAM. I am not an embedded Linux expert, but this does seem like a lot after tinkering with Raspberry Pi Pico and such. At the same time, it makes sense, since we do have a Node.js app running on the host and JavaScript isn't exactly a memory-efficient language.

The app itself is quite a sizeable codebase. Despite the fact that the company that made the device and software for it is already dissolved, I don't want to risk being sued over releasing the source code. I am pretty sure nobody would actually care, but I still want nothing to do with it. So instead we will just look at some file lists.

Top-level directory structure looks like this:

```
root@nt-core:/srv/server# ls -la
drwxr-xr-x   13 nodejs   nogroup       4096 Oct 23 10:47 .
drwxr-xr-x    3 root     root          4096 Oct 14 12:13 ..
-rw-r--r--    1 nodejs   nogroup       8125 Oct 14 12:13 Gruntfile.js
-rw-r--r--    1 nodejs   nogroup       1916 Oct 14 12:13 app.js
drwxr-xr-x    2 nodejs   nogroup       4096 Oct 14 12:13 app_data
drwxr-xr-x   10 nodejs   nogroup       4096 Oct 14 12:13 bin
-rw-r--r--    1 nodejs   nogroup       1278 Oct 14 12:13 bower.json
```

```
drwxr-xr-x    2 nodejs   nogroup       4096 Oct 23 10:40 info
-rw-r--r--    1 nodejs   nogroup         16 Oct 14 12:13 jira_version
-rw-r--r--    1 nodejs   nogroup       2673 Oct 14 12:13 karma.conf.js
drwxr-xr-x    2 nodejs   dialout       4096 Oct 23 10:49 logs
drwxr-xr-x    4 nodejs   nogroup       4096 Oct 14 12:13 modules
drwxr-xr-x   17 nodejs   nogroup       4096 Oct 14 12:13 node_modules
-rw-r--r--    1 root     root             0 Oct 23 10:40 nodejs.log
-rw-r--r--    1 nodejs   nogroup      76023 Oct 14 12:13 npm-shrinkwrap.jso
-rw-r--r--    1 nodejs   nogroup       2216 Oct 14 12:13 package.json
drwxr-xr-x    6 nodejs   nogroup       4096 Oct 14 12:13 production
drwxr-xr-x    6 nodejs   nogroup       4096 Oct 14 12:13 public
drwxr-xr-x    4 nodejs   nogroup       4096 Oct 14 12:13 routes
drwxr-xr-x    3 nodejs   nogroup       4096 Oct 14 12:13 scripts
drwxr-xr-x    4 nodejs   nogroup       4096 Oct 14 12:13 views
```

From what I could figure out, the app consists of two parts. The first part is responsible for actually reading usage data from the electricity meter connected to the device. This part is called a “Pulse app” and its binary is located in the `bin` folder:

```
root@nt-core:/srv/server# ls -la bin
drwxr-xr-x   10 nodejs   nogroup       4096 Oct 14 12:13 .
drwxr-xr-x   13 nodejs   nogroup       4096 Oct 23 10:47 ..
drwxrw----    2 nodejs   dialout       4096 Nov 16 00:12 aggregate
drwxrw----    2 nodejs   dialout       4096 Oct 23 11:03 cfg
-rwxr-xr-x    1 nodejs   dialout      52418 Oct 14 12:13 ct-read-daemon
drwxrw----    2 nodejs   dialout       4096 Nov 16 01:00 daily
-rwxr-xr-x    1 nodejs   nogroup       1155 Oct 14 12:13 get_sys_versions.s
drwxrw----    2 nodejs   dialout     118784 Nov 16 03:00 hourly
drwxrw----    2 nodejs   dialout       4096 Nov  1 01:00 monthly
drwxr-xr-x    2 nodejs   nogroup       4096 Oct 14 12:13 output
-rwxr-xr-x    1 nodejs   dialout      34543 Oct 14 12:13 pulse-app
-rwxr-xr-x    1 nodejs   dialout     117029 Oct 14 12:13 pulse.ko
-rwxr-xr-x    1 nodejs   nogroup        758 Oct 14 12:13 reset_nrg_mgr.sh
drwxrw----    2 nodejs   dialout       4096 Nov 16 01:00 weekly
drwxrw----    2 nodejs   dialout       4096 Oct 23 11:00 yearly
```

Judging from the debug symbols, my initial guess was that this is a regular C appication, which seems appropriate for the task of interacting with low-

level GPIO pins. However, `pulse.ko` file got me interested. `.ko` extension usually means “Kernel Object”, which would suggest that this is actually a kernel module. I do not know a first thing about kernel modules, so I might be wrong here.

Pulse app reads the data from the GPIO pins and stores the results in CSV files. These CSV files are split by month, day and hour in the directories with the respective names, still in the `bin` folder. Such separation is no coincedence. Historical data view in the web UI of the energy manager supports displaying the data only by month, by day and by hour.

Along with the Pulse app, there is the second part of the application. A Node.js app reads CSV files populated with energy usage data and displays them to the user in the web UI. It uses Node.js 0.10.26, Express.js 4.13.3 and Socket.io 1.3.6.

Scrolling through the dependencies, I noticed an `mqtt` package. This was intriguing, because I did not see any message broker interaction until now. After reading the sources for a little while, this seemed to be an unfinished cloud integration that Netthings promised in their brochure. There are even some hardcoded IPs mentioned in the source, which are used to connect to a message broker. Unsurprisingly, none of them are up any more. I am not even sure how that would work, since the device does not have an internet access.

All in all, this was a very fun investigation! I grew accustomed to calling it “my urban archeology project”. I now have access to one of the weirdest Edge Computing platforms imaginable. If you have any fun ideas on what to do with it, feel free to drop me a line at hi@laplab.me!

P.S. As a final touch, I decided to leave a small note in the home directory. It’s kind of weird to realise that I am probably the only person who would actually read it. But maybe in some distant future another software engineer will live in this apartment and discover it. Time will tell :)

```
root@nt-core:~# cat hello_stranger.txt
Hello, stranger!

My name is Nikita Lapkov and I am the person who reset the root
password on this device to an empty string. You are welcome!

I wrote an article about it, you can read it here:
https://laplab.me/posts/whats-that-touchscreen-in-my-room/

Feel free to explore and have fun with this little device.

Peace!
root@nt-core:~#
```