(Reverse Engineering the Firmware Binary)

Manual-Extraction, Analysis, Modification and Packing of Router Firmware

Contents:

Background: In the world of ever-increasing interconnection of computing, mobile and embedded devices, their security has be-come critical. The security of embedded devices and their firmwares is the new trend in the embedded market. The security requirements and expectations for computingdevices are being constantly raised as the world moves to-wards the Internet of Things. This is especially true for embedded devices and their software counterpart – firmwares– which is also their weakest point.

IoT device Firmware Reverse Engineering: It is a process to understand the device architecture, functionality and vulnerabilities present in the device incorporating different methods.

Firmware: Piece of code written for specific hardware to perform different operations and control the device.

Manual-Extraction and Packing means (in context of our article): By manual-extraction and packing we mean to extract and pack the target firmware by using command line tool like “dd” or through some program like python script that is without using any automated tools like “binwalk” and “Firmware-mod-kit” because sometimes these automated tools fails or we have some specific requirement. In such type of situations, it is very handy to build or write your own tools/scripts as per your specific need and requirement.

Problem statement: We have to modify the router firmware binary. We used the automated tool “Firmare-Mod-Kit” for its extraction, analyse it, modify as per our need (Bricking) and again try to pack it through the same tool “fmk” but it fails, even if we simply extract the firmware through “fmk” and again pack it without doing any change, “fmk” still fails. So here at this point there was a requirement that how we can extract & pack this target firmware successfully which leads us towards manual extraction of firmware.

Objective: Our objective is to modify/customised the router firmware binary. The target device is a router which is running an embedded Linux OS. Routers are the main incoming and outgoing points to the outside world on a computer network, and as such are a prime location for sniffing traffic and performing man in the middle (MITM) attacks. If we control the router then we controls the network traffic.

We reverse engineer targeted firmware binary of the Router and modified it so that when upload to the router, the router becomes unusable(Bricked):

Strategy Followed:

- The firmware for the device can be downloaded from the vendor's website.
- Get as much information about the firmware by using tools like file, readelf, binwalk etc. and from online sources regarding file type, Boot Loader and its version, file system, Architecture, Version, Kernel version, compression used or any encryption used etc.
- File system, header and kernel are extracted by using a python script.
- Analyse the file system as the file system includes default settings and binaries from the device which then can then be reverse engineered for potential exploits.The purpose of this is to identify the architecture information of the device, as well as finding vulnerabilities and other potential avenues of attack.
- Modify or change the firmware's file system as per our requirement.
- Again pack the firmware manually by using our python script.

Tools Used:

- **File tool**
- **Binwalk**
- **Readelf**
- **Python Scrpit (maual_extract.py) for extraction and packing of firmware.**
- **Unsquashfs**
- **mksquashfs**
- **Firmware-analysis-Tool-Kit**

Demonstration of extracting, analysing and modifying the firmware binary manually:

Figure: 1

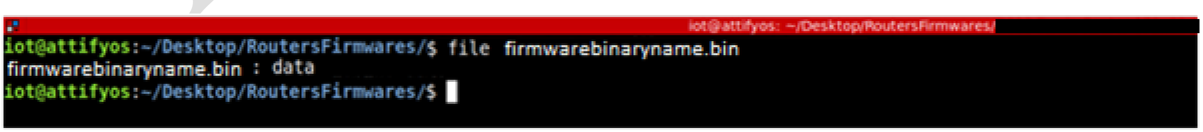
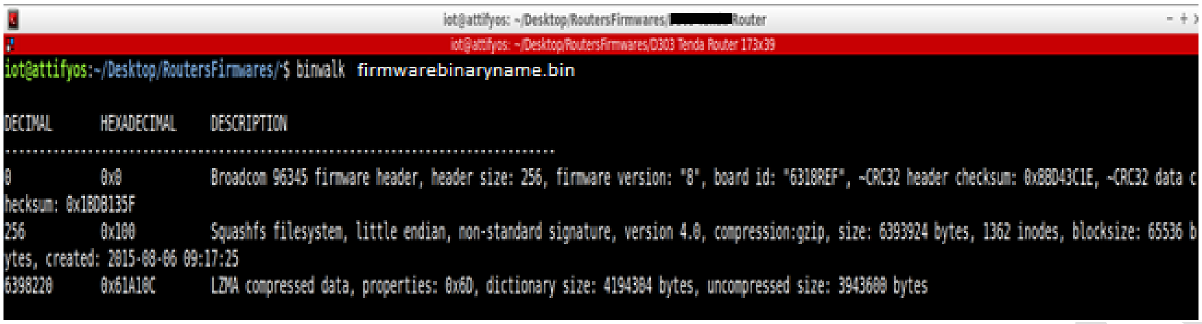


Figure 1 above shows the output of a tool called file, this tool attempts to classify a fileand tests if the file is in a certain format. Here it simply tells that the file type is data. We also ran the file tool on an extracted executable from the bin directory in the filesystem later on.

Figure2:



In Figure2 above, we have used "Binwalk" which is an open-source tool for analyzing, reverse engineering and extracting firmware images. Binwalk is able to scan a firmware image and search for file signatures to identify and extract filesystem images, executable code, compressed archives, bootloader and kernel images, file formats like JPEGs and PDFs, and many more!

Both *file* and *binwalk* tools use the *libmagic* library to identify file signatures. But binwalk additionally supports a list of custom magic signatures to find compressed/archived files, firmware headers, Linux kernels, bootloaders, filesystems, and so on!

Binwalk has several options you can use but here we only simply use binwalk without any option to get useful information about our target firmware image. Binwalk structure has three sections:

1. File location or offset in decimal format
2. File location or offset in hexadecimal format
3. Description of what was found at that location

Points to be noted from above information: The image mentions that the firmware belongs to chipset Broadcom 96345 and also firmware version 8 in first line. And based on the image found at the address 0x100, we can see that the rootfs is a squashfs filesystem in little endian format. Other useful information like gzip compression and size is also shown. Further in last line at address 0x61A10C shows that lzma compressed data and is most probably be the kernel.

The information provided above is very useful, especially the offset/addresses information regarding each part because we will going to use these offsets in our python script for manual extraction to tell the script that from which offset to extract these different parts.

Figure 3:

```
#!/usr/bin/env python3

import sys

class FirmwarePart:
  def __init__(self, name, offset, size):
    self.name = name
    self.offset = offset
    self.size = size
```

Starting address or offset

Total size of the part

```
firmware_parts = [
  FirmwarePart("broadcom_header", 0x0, 0x100),
  FirmwarePart("squashfs", 0x100, 0x61A00C),
  FirmwarePart("kernel.lzma", 0x61A10C, 7,734,505-0x61A10C),
]

if sys.argv[1] == "unpack":
  f = open(sys.argv[2], "rb")
  for part in firmware_parts:
    outfile = open(part.name, "wb")
    f.seek(part.offset, 0)
    data = f.read(part.size)
    outfile.write(data)
    outfile.close()
    print(f"Wrote {part.name} - {hex(len(data))} bytes")
elif sys.argv[1] == "pack":
  f = open(sys.argv[2], "wb")
  for part in firmware_parts[0:]:
    i = open(part.name, "rb")
    data = i.read()
    f.write(data)
    padding = (part.size - len(data))
    print(f"Wrote {part.name} - {hex(len(data))} bytes")
```

Above is the code for our python script which we used to extract and pack the firmware. In the script above we have defined a list "firmware parts" under class FirmwarePart, in which we mention the part name, its offset and total size of particular part name to pack or unpack each part as per the argument ("pack or "unpack") we passed at run time to run the script as mentioned below:

python3 ourscriptname.py argument(pack or unpack) firmwarefilename

Example:

To unpack or extract the firmware

Python3 manual-script.py unpack OriginalFirmwareBinary.bin

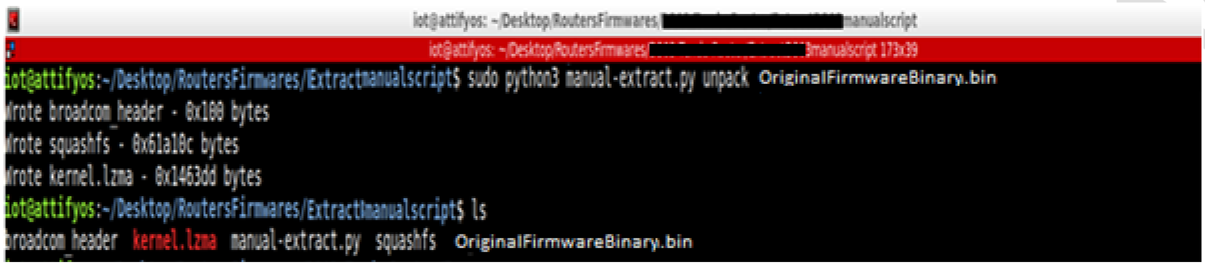
To pack the firmware:

\# **Python3** **manual-script.py** **pack** **NewFirmware.bin**

Manual Extraction and Packing:

Step1: Extraction of firmware through python script “manual-script.py”

\# **Python3** **manual-script.py** **unpack** **OriginalFirmwareBinary.bin**

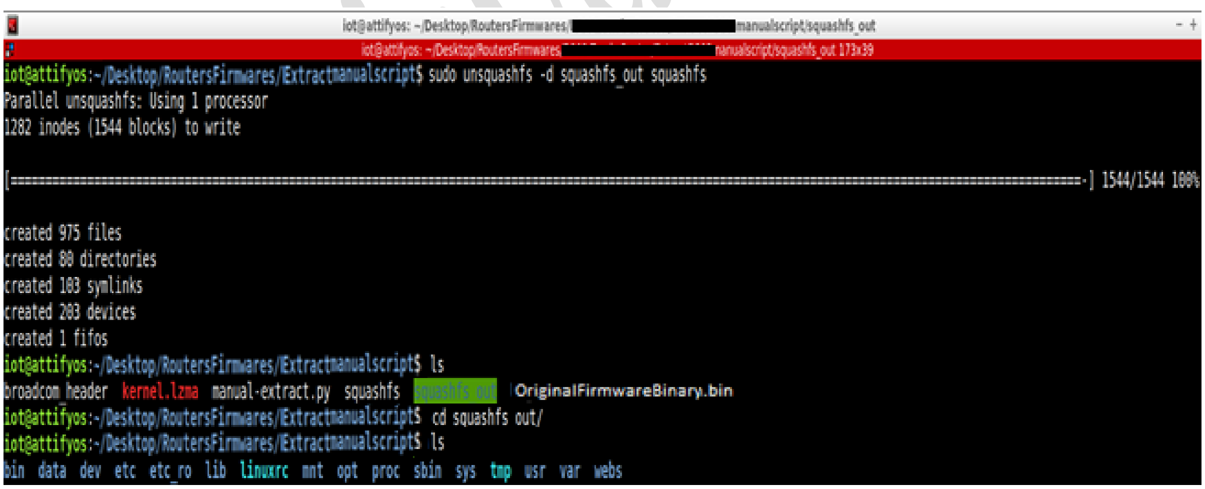


```
iot@attifyos: ~/Desktop/RoutersFirmwares/[redacted]manualscript
iot@attifyos: ~/Desktop/RoutersFirmwares/[redacted]manualscript 173x39
iot@attifyos:~/Desktop/RoutersFirmwares/Extractmanualscript$ sudo python3 manual-extract.py unpack OriginalFirmwareBinary.bin
Wrote broadcom_header - 0x100 bytes
Wrote squashfs - 0x61a10c bytes
Wrote kernel.lzma - 0x1463dd bytes
iot@attifyos:~/Desktop/RoutersFirmwares/Extractmanualscript$ ls
broadcom_header  kernel.lzma  manual-extract.py  squashfs  OriginalFirmwareBinary.bin
```

After extraction as shown in above figure, three parts of our firmware image is extracted with name as Broadcom_header, squashfs, kernel.lzma. Currently our interest is in file system, so we unpack squashs with “unsquashfs” utility to get complete file system as shown below in Figure4:

Step2: Unpack squashs with “unsquashfs” utility to get complete file system

Figure 4:



```
iot@attifyos: ~/Desktop/RoutersFirmwares/[redacted]manualscript/squashfs_out
iot@attifyos: ~/Desktop/RoutersFirmwares/[redacted]manualscript/squashfs_out 173x39
iot@attifyos:~/Desktop/RoutersFirmwares/Extractmanualscript$ sudo unsquashfs -d squashfs_out squashfs
Parallel unsquashfs: Using 1 processor
1282 inodes (1544 blocks) to write

[==========================================================================================] 1544/1544 100%

created 975 files
created 80 directories
created 103 symlinks
created 203 devices
created 1 fifos
iot@attifyos:~/Desktop/RoutersFirmwares/Extractmanualscript$ ls
broadcom_header  kernel.lzma  manual-extract.py  squashfs  squashfs_out  OriginalFirmwareBinary.bin
iot@attifyos:~/Desktop/RoutersFirmwares/Extractmanualscript$ cd squashfs_out/
iot@attifyos:~/Desktop/RoutersFirmwares/Extractmanualscript$ ls
bin  data  dev  etc  etc_ro  lib  linuxrc  mnt  opt  proc  sbin  sys  tmp  usr  var  webs
```

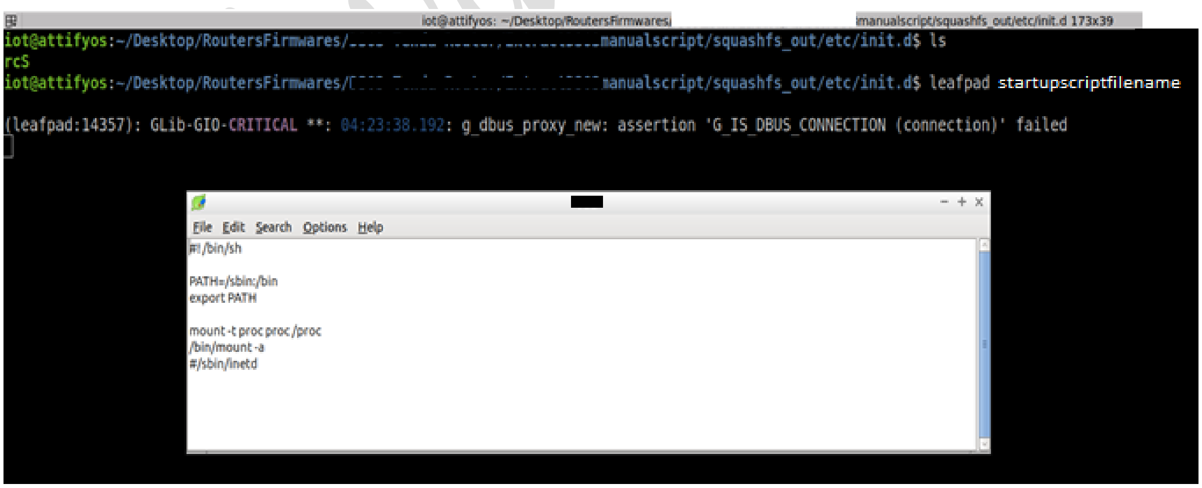
As shown above in figure 4, squashfs_out directory contains the complete filesystem of our firmware image which we can further analyse to find vulnerabilities and how we can modify it in our best interest.

Step3: Analysis of file system and modification.

Figure 5:

Now as shown in Figure 5, we traverse various directories like bin, etc and webs which contains important files which we further analyse to find any loopholes which either helps us in creating any backdoor or we can modify to brick the device. For example, there are some start-up scripts present in some directories which runs on start-up of the router and do various work like initialization and setting various parameters. We modify such files to achieve our objective.(The name and location of such files is not mentioned here intentionally).

Figure 6:



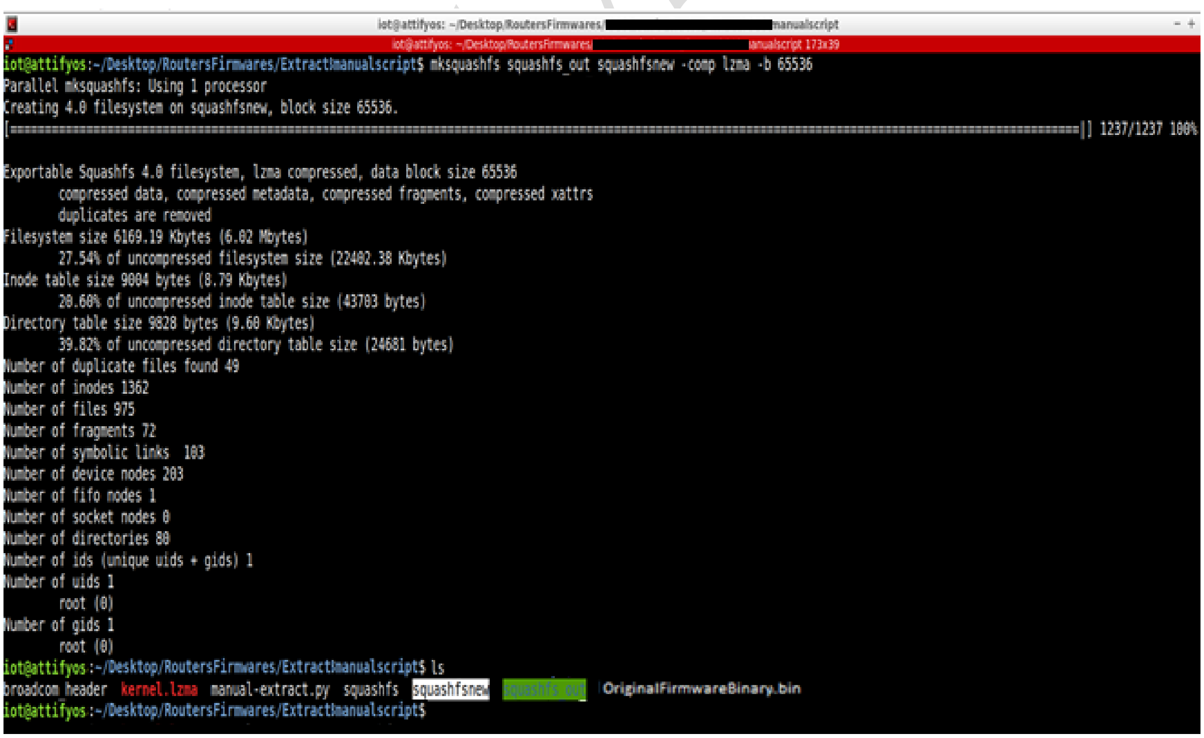
As shown above in Figure 6, we simply accessing the file and open it with leafpad. The contents of file are also shown below in Figure 7 also.

Figure 7:

As we know from above discussion that start-up script file is important and runs on router start-up and we also knew that the system is a linux system. We further analyse busybox file present in /bin directory to know its version, commands it supports by using file and emulation software qemu.We can as per our requirement can write our customised code or script and can run through it on system start-up. It is just one method, there are number of files available and we further explore them as per our need. Our malicious process will run in background.

Step4: Packing of squashfs file system using "mksquashfs" utility after modification.

Figure 8:



After modifying the required files in our extracted file system, we simply again pack the unpacked squashfs (squashfs_out) by using "mksquashfs" utility as shown above in Figure8.

Step5: Packing of or creating new-firmware image using our python script manual-extract.py.

Python3manual-script.pypackNewFirmware.bin

Figure 9:

Now at last step, we simply build our new modified firmware again by using our python script manual-extract.py as shown in figure 9 above. It successfully created and same as original firmware (check binwalk output above for new firmware file). We can now upload it on our target router and brick it.

Demonstration of our modified new-firmware through emulation.

To demonstrate that after modifying the firmware it does not work after uploading on router we emulate it through firmware-analysis-tool kit which is based on qemu and automate the whole process.

First we emulate our original firmware file as shown in Figure 10 and Figure 11 and finds that it works fine. We able to emulate it and also accessing its file system (Figure 11). This confirms us that the original firmware is working fine.

Figure 10:

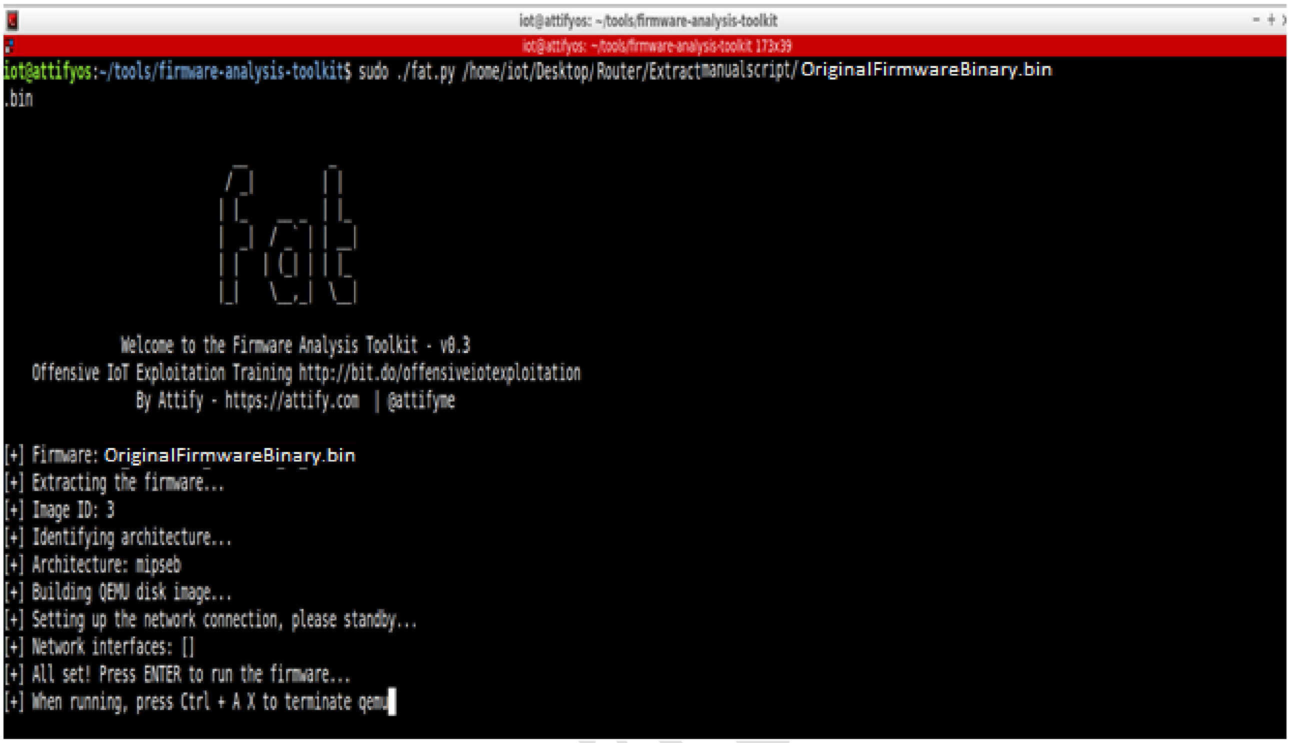
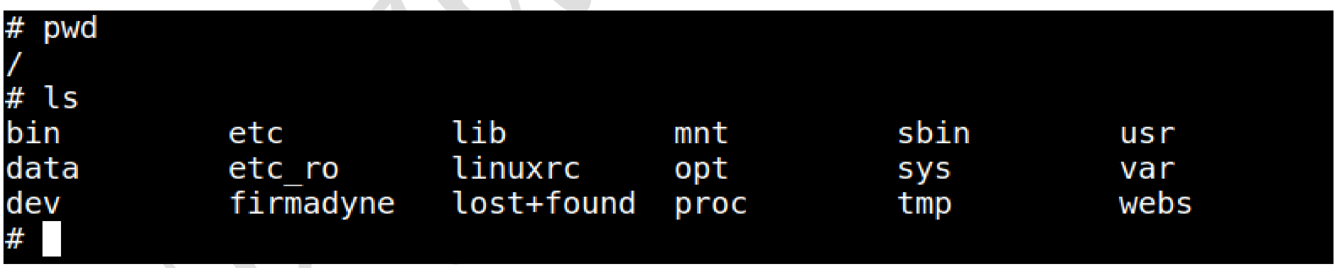
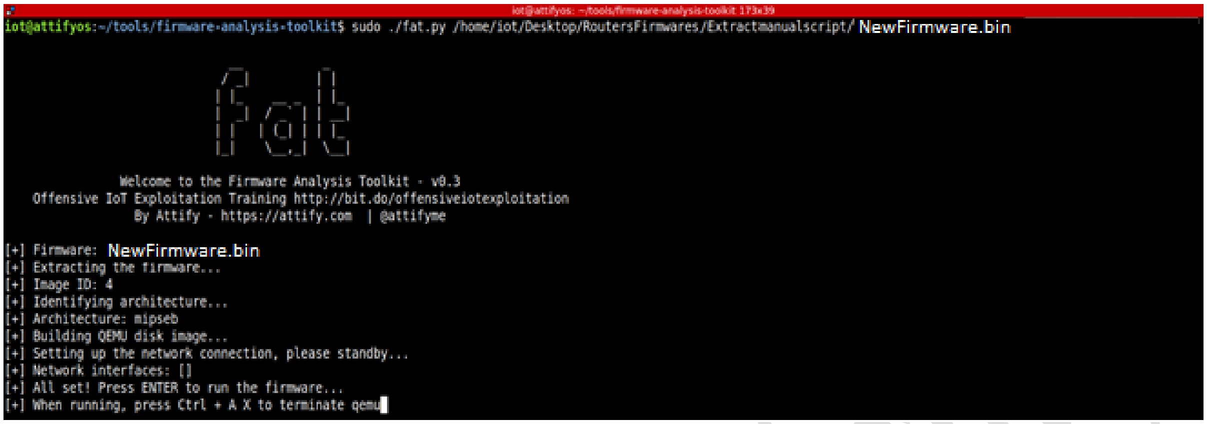


Figure 11:



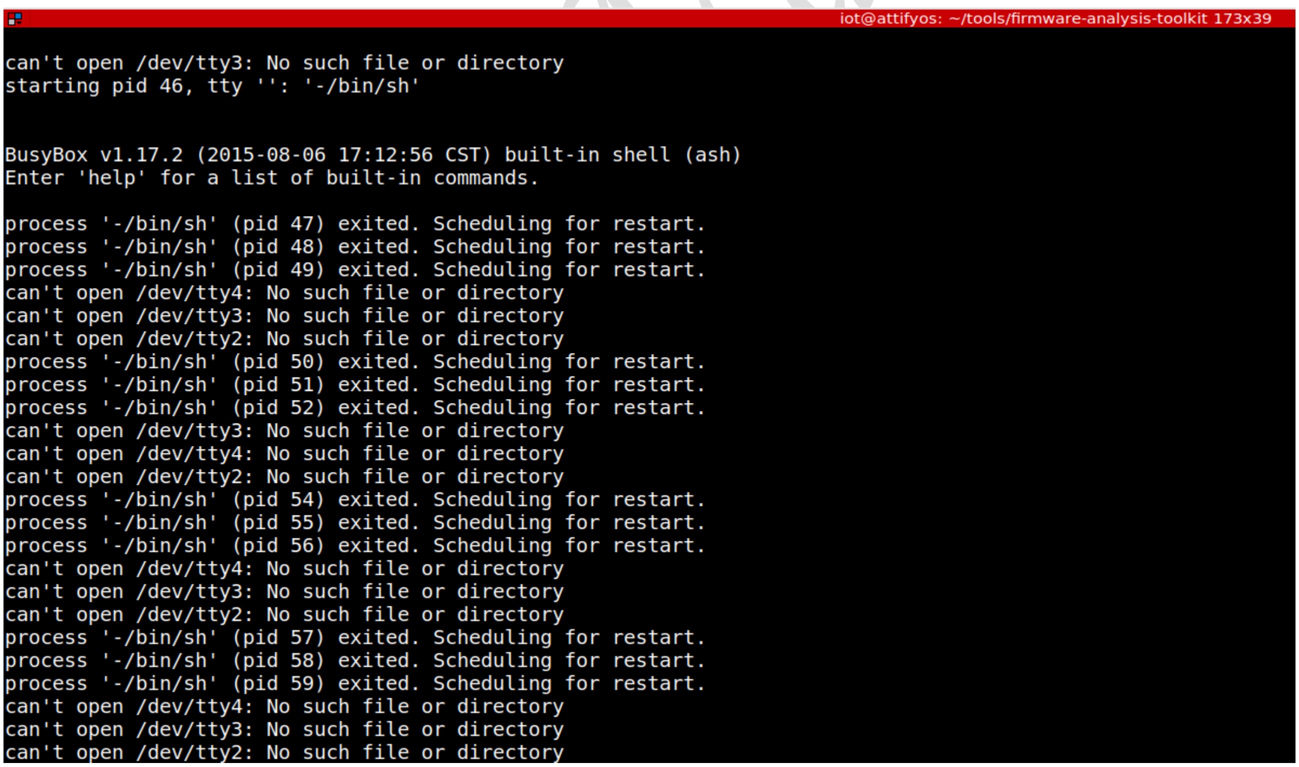
Now our next task is to emulate our modified new-firmware and check whether we can be able to brick the router or not.

Figure 12:



As shown in Figure 12, we are now emulating our modified new-firmware file with firmware-mod-kit. As shown in Figure 13 below that emulation is failed and router is restarting again and again. Our purpose and objective is fulfilled.

Figure 13:



Conclusion:

Here, we have analysed, extracted the entire filesystem of the target Firmware by using python script. We further modified the firmware and check through emulation that it did not work after modification.

Note: **Here in the report, the name of firmware binary, its version, firmware vendor name and start-up scripts files name and location are not mentioned intentionally for the purpose.**