Firmware Analysis & Filesystem Extraction

redfoxsec.com/blog/analyzing-firmware-and-extracting-filesystem

Kunal Kumar June 4, 2024

Analyzing Firmware and Extracting Filesystem

- June 4, 2024
- Active Directory
- Kunal Kumar

Every modern device—whether it's a Wi-Fi router, smart TV, medical equipment, or industrial controller—relies on firmware. Firmware is the invisible layer of code that powers hardware, managing communication between components, processing data, and enabling functionality.

Unlike regular software, firmware runs deep inside devices, yet it's often overlooked during security assessments. This makes it a prime target for attackers. Hidden inside firmware images, you might find hardcoded passwords, outdated libraries, misconfigurations, or even backdoors. A successful firmware compromise can give adversaries persistent, stealthy access—often invisible to traditional security tools.

That's why learning to analyze and extract firmware is more than just reverse engineering—it's an essential skill for cybersecurity professionals. By inspecting firmware, you can uncover vulnerabilities before they're exploited, harden devices against attacks, and gain insights into how embedded systems operate.

This guide walks through the end-to-end process: acquiring firmware, extracting its filesystem, performing analysis, and interpreting findings. Whether you're a pentester, security researcher, or simply curious about how your devices work, this breakdown will help you explore the hidden world of firmware.

But here's the catch: firmware is often overlooked in security assessments, even though it's a prime target for attackers. Hidden within firmware images could be **hardcoded passwords, outdated libraries, misconfigurations, or even backdoors** that jeopardize an entire system. Once compromised, firmware can allow attackers to gain persistent, stealthy access—undetectable by traditional security measures.

That's why analyzing and extracting firmware isn't just an exercise in reverse engineering—it's **a critical skill for anyone serious about cybersecurity**. By peering inside, we can uncover vulnerabilities before adversaries exploit them, strengthen device security, and gain valuable insights into how embedded systems operate.

This guide walks you through the **end-to-end process of firmware analysis**, from acquisition to filesystem exploration, static and dynamic analysis, and finally, reporting findings. Whether you're a security researcher, pentester, or simply curious about what powers your devices, this step-by-step breakdown will show you how to crack open the black box of firmware and uncover its secrets.

Why Firmware Analysis Matters

Firmware analysis provides clear benefits for both individuals and organizations:

- **Discover vulnerabilities** like hardcoded credentials and outdated libraries.
- **Identify sensitive data exposures** that could compromise devices or user privacy.
- **Strengthen defenses** by detecting weaknesses before attackers exploit them.
- **Enable testing without hardware**, simulating penetration testing environments.
- **Gain operational insights** into device functionality at the lowest level.

In short, firmware analysis improves security posture and offers a cost-effective way to manage risks.

The Process of Firmware Extraction And Analysis

Step 1: Firmware Acquisition

The first step is obtaining a firmware image:

- Download directly from the device or the manufacturer's official site.
- Verify authenticity to ensure the image hasn't been tampered with.

Example:
For this walkthrough, we'll use `FW_WRT1900ACSV2_2.0.3.201002_prod`. Extracting it with Linux's `7z` tool yields raw firmware files for analysis.

```
$ git clone https://github.com/prokunal/Dumping-Router-Firmware-Image
```

Extracting the firmware image from the archive.

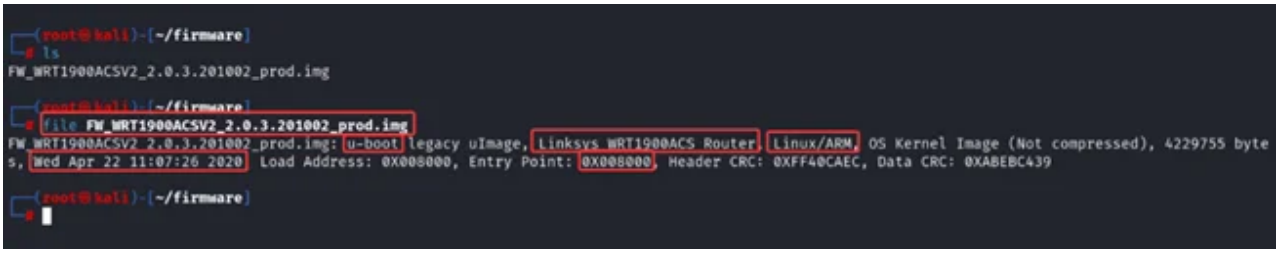
```
$ 7z x FW_WRT1900ACSV2_2.0.3.201002_prod.zip
```

Step 2: Firmware Extraction

With the image in hand, several tools help extract and analyze its content:

```
$ file FW_WRT1900ACSV2_2.0.3.201002_prod.img
```

file command – Provides metadata (architecture, compression type, creation date, entry point).



strings command – Reveals readable text within the image, often containing useful details.

```
$ strings FW_WRT1900ACSV2_2.0.3.201002_prod.img
```

From the above command, we can analyse the strings to find more details about the firmware.

binwalk – Identifies file signatures and extracts embedded data, such as compressed filesystems or hidden resources.

Binwalk is a versatile tool for identifying well-known file signatures within a given file. While its primary function is to analyse file structures, it can also be utilized in areas like Steganography. For instance, it can uncover hidden files within images.

In the context of router analysis, Binwalk proves invaluable for extracting the filesystem, allowing for deeper inspection and potential modification.

```
$ binwalk -e FW_WRT1900ACSV2_2.0.3.201002_prod.img –run-as=root
```

The above command output also gives basic information about the firmware and filesystem, such as image size, OS, CPU, product name, and compression type. The filesystem is JFFS2 in little-endian format.

Analysing the extracted file, found two files one is a jffs2 filesystem, and another one looks like a compressed file.

```
┌──(root㉿kali)-[~/firmware]
└─# ls
FW_WRT1900ACSV2_2.0.3.201002_prod.img  _FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted

┌──(root㉿kali)-[~/firmware]
└─# cd _FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted

┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted]
└─# ls
600000.jffs2  6870

┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted]
└─# du -sh *
8.0M    6870
27M     600000.jffs2

┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted]
└─# file 6870
6870: data

┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted]
└─#
```

Analysing the compressed file 6870 using strings to find more information about this file.

```
$ strings 6870
```

```
net/ipv6/ip6_offload.c
net/ipv6/inet6_hashtables.c
sk       RefCnt Type Proto  Iface R Rmem   User   Inode
%pK %-6d %-4d %04x   %-5d %1d %-6u %-6u %-6lu
6net→packet.sklist_lock
net/packet/af_packet.c
TPACKET version not supported.
3packet size is too long (%d > %d)
3packet size is too short (%d < %d)
3Packet exceed the number of skb frags(%lu)
6po→pg_vec_lock
Tx-ring is not supported.
3packet_mmap: vma is busy: %d
3Attempt to release alive packet socket: %p
TPACKET version not supported
3bridge: can't register sap for STP
net/bridge/br_fdb.c
4%s: adding interface %s with same address as a received packet
bridge_fdb_cache
4%s: received packet on %s with own address as source address
6bridge: RTM_NEWNEIGH with invalid state %#x
6bridge: RTM_NEWNEIGH with invalid vlan
6bridge: RTM_NEWNEIGH with invalid vlan id %d
6bridge: RTM_NEWNEIGH %s not a bridge port
6bridge: RTM_NEWNEIGH with unconfigured vlan %d on port %s
6bridge: RTM_DELNEIGH %s not a bridge port
6bridge: RTM_DELNEIGH with unconfigured vlan %d on port %s
plip
net/bridge/br_if.c
brport
failed insert local address bridge forwarding table
6%s: port %u(%s) entered %s state
5%s: root port %u not found for topology notice
6%s: topology change detected, %s
propagating
sending tcn bpdu
```

From the above command, it didn't give much information about the file.

Extracting the content of 6870 file using binwalk.

```
$ binwalk -e 6870 –run-as=root
```

```
┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted]
└─# binwalk -e 6870 --run-as=root

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
1904228       0x1D0E64        SHA256 hash constants, little endian
4112676       0x3EC124        SHA256 hash constants, little endian
5877920       0x59B0A0        Linux kernel version 3.10.3
6120324       0x5D6384        AES S-Box
6120580       0x5D6484        AES Inverse S-Box
6176102       0x5E3D66        Unix path: /var/run/rpcbind.sock
6261498       0x5F8AFA        MPEG transport stream data
6261758       0x5F8BFE        MPEG transport stream data
6902132       0x695174        Unix path: /dev/vc/0
6993884       0x6AB7DC        xz compressed data
7027944       0x6B3CE8        Unix path: /lib/firmware/updates/3.10.39
7048952       0x6B8EF8        Ubiquiti firmware additional data, name: UTE DEVICE DIAGNOSTIC, size: 1313820672 bytes, size2: 1179407699 bytes, CR
C32: 0
7194599       0x6DC7E7        Copyright string: "Copyright 2005-2007 Rodolfo Giometti <giometti@linux.it>"
7218153       0x6E23E9        Copyright string: "Copyright(c) Pierre Ossman"
7301128       0x6F6808        Unix path: /etc/init.d/ipv6_react_to_ra.sh
7304520       0x6F7548        Neighborly text, "NeighborSolicits6InDatagrams"
7304540       0x6F755C        Neighborly text, "NeighborAdvertisementsorts"
7309086       0x6F871E        Neighborly text, "neighbor %.2x%.2x.%pM lost rename link %s to %s"
7946423       0x7940B7        LZMA compressed data, properties: 0xC0, dictionary size: 0 bytes, uncompressed size: 64 bytes
7970360       0x799E38        ASCII cpio archive (SVR4 with no CRC), file name: "dev", file name length: "0x00000004", file size: "0x00000000"
7970476       0x799EAC        ASCII cpio archive (SVR4 with no CRC), file name: "dev/console", file name length: "0x0000000C", file size: "0x0000
0000"
7970600       0x799F28        ASCII cpio archive (SVR4 with no CRC), file name: "root", file name length: "0x00000005", file size: "0x00000000"
7970716       0x799F9C        ASCII cpio archive (SVR4 with no CRC), file name: "TRAILER!!!", file name length: "0x0000000B", file size: "0x00000
000"
7996352       0x7A03C0        CRC32 polynomial table, little endian
8075823       0x7B3A2F        LZMA compressed data, properties: 0xC0, dictionary size: 0 bytes, uncompressed size: 32 bytes
8227376       0x7D8A30        Copyright string: "Copyright 2000-2013, Marvell International Ltd."
8275455       0x7E45FF        LZMA compressed data, properties: 0xC0, dictionary size: 0 bytes, uncompressed size: 192 bytes
```

From the above output, we can assume that this is an updated file of the firmware for version 3.10.39.

Analysing the extracted 6870 files, we found that they don't contain much informational data on it, except 799E38.cpio file which might contain interesting information.

```
┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted]
└─# ls
600000.jffs2  6870  _6870.extracted

┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted]
└─# cd _6870.extracted

┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted/_6870.extracted]
└─# ls
6AB7DC.xz  7940B7  7940B7.7z  799E38.cpio  7B3A2F  7B3A2F.7z  7E45FF  7E45FF.7z  console  cpio-root  dev  root

┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted/_6870.extracted]
└─#
```

Analysing 799E38.cpio file using strings command.

Cpio stands for "copy in, copy out". It is used for processing archive files such as *.cpio or *.tar and can transfer files in both directions from or into them. Additionally, this command allows accessing archived folders by copying files directly.

```
$ strings 799E38.cpio
```

```
mark
Linux
Shelby
3.10.39
#1 SMP Wed Apr 22 04:07:09 PDT 2020
(none)
swapper
atst
earlycon
marvell,orion-system-controller
marvell,armada-370-xp-system-controller
marvell,armada-375-system-controller
marvell,armada-380-system-controller
arm,cortex-a9-scu
arm,cortex-a9-scu
marvell,coherency-fabric
marvell,armada-370-coherency-fabric
marvell,armada-375-coherency-fabric
marvell,armada-380-coherency-fabric
marvell,armada-370-pmsu
marvell,armada-370-xp-pmsu
marvell,armada-380-pmsu
marvell,armada-370-cpu-reset
marvell,armada-xp-cpu-reset
marvell,armada-375-cpu-reset
marvell,armada-380-cpu-reset
marvell,armada-375-usb-cluster
marvell,armada-380-usb-utmi
marvell,armada-375-common-phy-configuration
marvell,armada-375-serdes-pipe-configuration
marvell,armada-375-ip-configuration
/sbin/poweroff
/sbin/modprobe
atst
tasks
cgroup.procs
```

Analysing 600000.jffs2 file and examining its content.

Using file command to analyse file type.

```
$ file 600000.jffs2
```

```
┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted]
└─# ls
600000.jffs2  6870  _6870.extracted

┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted]
└─# file 600000.jffs2
600000.jffs2: Linux jffs2 filesystem data little endian

┌──(root㉿kali)-[~/firmware/_FW_WRT1900ACSV2_2.0.3.201002_prod.img.extracted]
└─#
```

From the above output, it shows the filesystem is jffs2 in little endian format.

Journalling Flash File System version 2 or JFFS2 is a log-structured file system for use with flash memory devices.

Mounting the JFFS2 filesystem.

1. Create the block device.

```
$ mknod /dev/mtdblock0 b 31 0
```

2. Create a directory for the **JFFS2 filesystem** in /mnt directory.

```
$ mkdir /mnt/jffs2_file
```

3. Load the required **kernel modules**. These modules provide the necessary functionality to interact with JFFS2 filesystems and memory technology devices.

```
$ modprobe jffs2
$ modprobe mtdram
$ modprobe mtdblock
```

4. Write the image to **/dev/mtdblock0** using **dd too.**

```
$ dd if=600000.jffs2 of=/dev/mtdblock0
```

5. Mount the filesystem to **/mnt/jffs2_file**

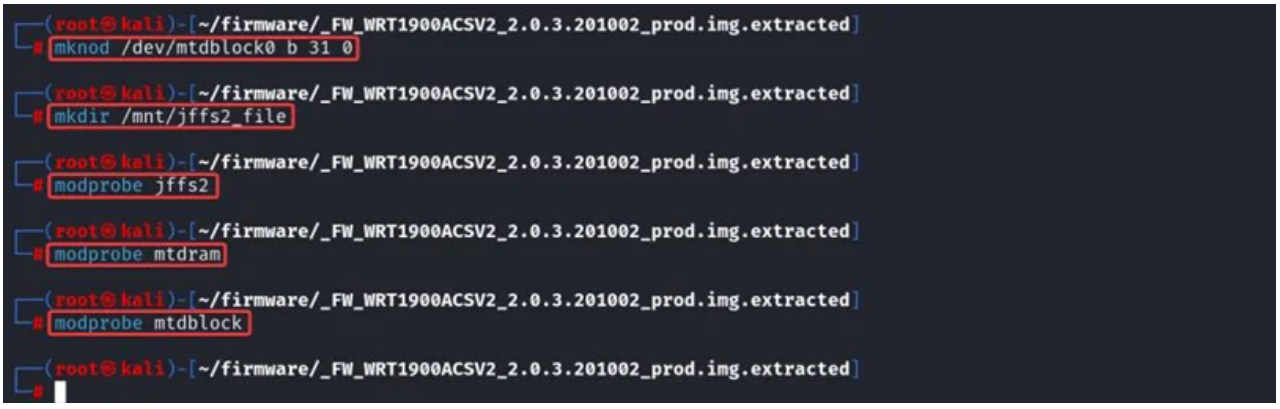
```
$ mount -f jffs2 /dev/mtdblock0 /mnt/jffs2_file
```

6. Jump into the mounted filesystem.

```
$ cd /mnt/jfffs2_file
```

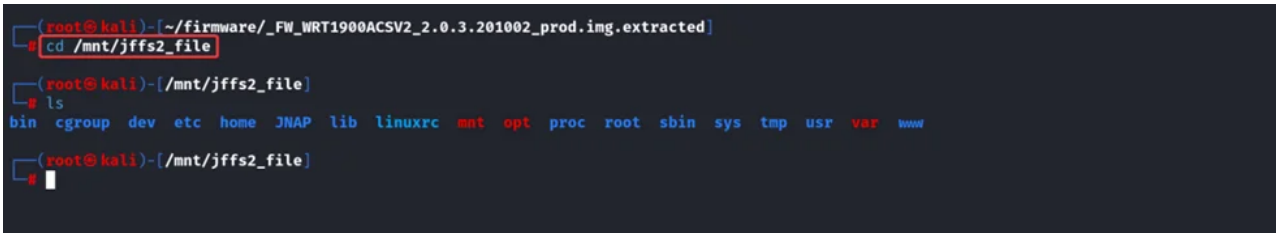
This command sequence facilitates the extraction and mounting of a JFFS2 filesystem from a router firmware image.

1. It initializes a block device **mtdblock** for accessing flash memory.
2. Set up a directory to mount the **JFFS2 filesystem**.
3. Necessary kernel modules (jffs2, mtdram, mtdblock) are loaded to enable interaction with the JFFS2 filesystem.
4. The firmware image is written onto the block size.
5. The JFFS2 filesystem is mounted onto the designated directory.



Step 3: File System Analysis

Moving to /mnt/jffs2_file, we can list our mounted file.



Analysing the **bin** folder, found out it is using busy box. In this firmware, most of the binary is linked to a busy box which is explained below; looking for other binaries, we can assume that it must be using services like SQLite3, smb, and http.

Understanding Busy Box

BusyBox is often referred to as the *"**Swiss Army Knife of Embedded Linux**."* It combines lightweight, stripped-down versions of many common UNIX utilities into a single executable file. Instead of having dozens of separate binaries for commands like `ls`, `cat`, `cp`, `mv`, `grep`, **or** `tar`, BusyBox provides them all within one compact program.

This design makes it ideal for **embedded systems** such as routers, IoT devices, and consumer electronics, where storage and memory are limited. While each tool included in BusyBox offers fewer features than its full-sized counterpart in standard Linux distributions, they retain enough functionality to handle most essential tasks.

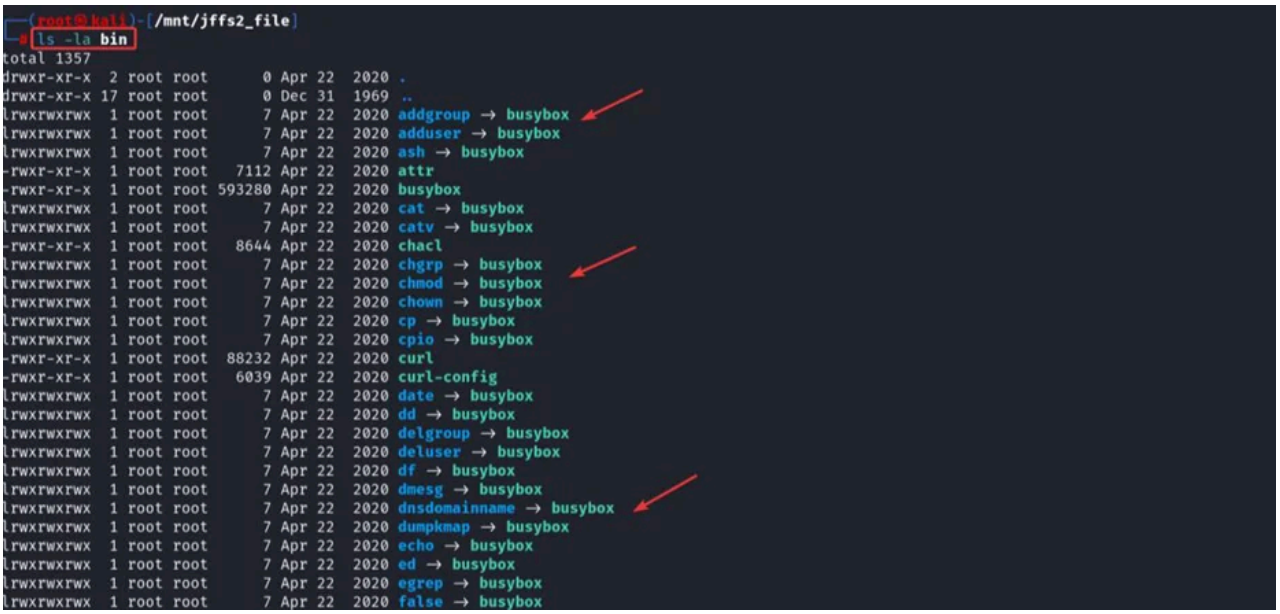
Key benefits of BusyBox include:

- **Efficiency** – Reduces storage and memory footprint by consolidating utilities into one binary.
- **Flexibility** – Can be configured at build time to include only the commands needed for a specific device.
- **Portability** – Widely adopted across embedded Linux systems, ensuring consistency and reliability.
- **Simplicity** – Provides a minimal yet functional command-line environment, perfect for troubleshooting and scripting.

Common uses in firmware analysis:
When examining an extracted firmware's filesystem, finding that many binaries link to BusyBox is a strong indicator the device is running a Linux-based embedded system. Analysts can then infer:

- The system likely uses BusyBox for core shell operations.
- Configuration files may reveal which services (like `httpd`, `telnetd`, or `udhcpc`) are being managed by BusyBox.
- BusyBox's built-in tools can reveal much about how the device is initialized and controlled.

In short, BusyBox is a cornerstone of embedded Linux environments: small, versatile, and highly optimized for constrained hardware.

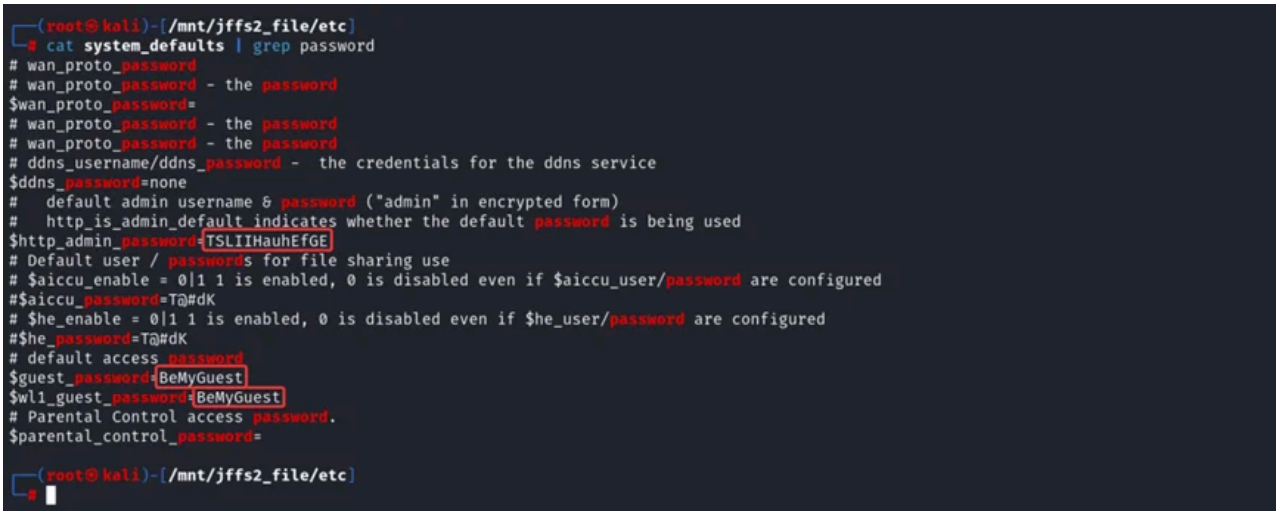


Step 4: Static Analysis

Examine **etc** folder for configuration files.

Analysing the etc folder, we can find many interesting configuration files related to the router. Analyzing the system_defaults file contains juicy information and reveals the password for http_admin_password and many others.

```
$ cat system_defaults | grep password
```



We can analyse all configuration files to find more information about the product. Some of them which store interesting information about the product are **buildate**, **version**, **dropbear_rsa_host_key.**

TL; DR

Analyzing firmware and extracting filesystems is a powerful way to understand and secure embedded systems. Through this process, we can:

- Expose vulnerabilities.
- Identify misconfigurations.
- Gain architectural insights.

By systematically dissecting firmware, organizations can build stronger defenses, meet compliance standards, and stay ahead of evolving threats.

At **Redfox Security**, our global team of security consultants helps organizations identify vulnerabilities and strengthen their defenses. If you're looking to improve your security posture, **contact us** to discuss your testing needs.

We also offer **comprehensive courses** to help you build expertise in areas like firmware analysis and embedded security. Join us and start your journey toward mastering the hidden layers of technology.

PreviousCybersecurity Strategies For Small And Mid-Sized Businesses (SMBs)
NextSecurity Advisory – Multiple Vulnerabilities in Netgear WNR614 Router

Recent Blog

September 09, 2025
Is APK Decompilation Legal? What You Need To Know
September 06, 2025
When Hackers Hit the Road: The Jaguar Land Rover Cyberattack
September 05, 2025

[This Is the Hacker’s Swiss Army Knife. Have You Heard About It?]