Army of Undead – Tailored Firmware Emulation

#lockdownlivestream

Researched at SEC Consult

Case studies done at SEC Consult
About me.

Thomas Weber
t.weber@sec-consult.com

SEC Consult Unternehmensberatung GmbH
Leopold-Ungar-Platz 2/3/3
1190 Vienna, AUSTRIA

www.sec-consult.com
Outline

• What? Why?
• Nowadays Firmware Development
• Detection of Essential Parts
• Preparing Fake Images
• Demo Time
• Monitoring and Debugging
• Scaled Study
• Vulnerabilities
• Conclusion and Further Work
The talk is about …

- **Linux based** firmware with **EABI**, no complete RTOS like VXworks. Windows can be covered with a similar approach but was not implemented in this prototype.
The talk is about …

- **Linux based firmware** with **EABI**, no complete RTOS like VXworks. Windows can be covered with a similar approach but was not implemented in this prototype.
- dynamic analysis, which means: libraries, networked services and known exploits.
The talk is about …

- **Linux based** firmware with **EABI**, no complete RTOS like VXworks. Windows can be covered with a similar approach but was not implemented in this prototype.
- dynamic analysis, which means: libraries, networked services and known exploits.
- a feasibility study about a workflow to realize simple firmware emulation with open source tools. It is not a ready-to-use tool!
The talk is about …

- **Linux based** firmware with **EABI**, no complete RTOS like VXworks. Windows can be covered with a similar approach but was not implemented in this prototype.
- dynamic analysis, which means: libraries, networked services and known exploits.
- a feasibility study about a workflow to realize simple firmware emulation with open source tools. It is not a ready-to-use tool!
- more than 170 firmware images that were emulatable across more than 49 vendors.
The talk is about …

- **Linux based** firmware with **EABI**, no complete RTOS like VXworks. Windows can be covered with a similar approach but was not implemented in this prototype.
- dynamic analysis, which means: libraries, networked services and known exploits.
- a feasibility study about a workflow to realize simple firmware emulation with open source tools. It is not a ready-to-use tool!
- more than 170 firmware images that were emulatable across more than 49 vendors.
- advisories that have been released, based on emulated devices only.
The talk is about …

- **Linux based** firmware with **EABI**, no complete RTOS like VXworks. Windows can be covered with a similar approach but was not implemented in this prototype.

- dynamic analysis, which means: libraries, networked services and known exploits.

- a feasibility study about a workflow to realize simple firmware emulation with open source tools. It is not a ready-to-use tool!

- more than 170 firmware images that were emulatable across more than 49 vendors.

- advisories that have been released, based on emulated devices only.

- a project that was started in January 2017.
Because …

• it can be used to do remote testing with frame-conditions
Why? // Vantage Point

Because …

- it can be used to do remote testing with frame-conditions
- it was helpful for a lot of research/customer-projects
Why? // Vantage Point

Because …

- it can be used to do remote testing with frame-conditions
- it was helpful for a lot of research/customer-projects
- dynamic analysis is currently limited to few architectures like ARM and MIPS
Why? // Vantage Point

Because …

• it can be used to do remote testing with frame-conditions
• it was helpful for a lot of research/customer-projects
• dynamic analysis is currently limited to few architectures like ARM and MIPS
• static analysis is well covered by many (non-)commercial solutions
Because …

- it can be used to do remote testing with frame-conditions
- it was helpful for a lot of research/customer-projects
- dynamic analysis is currently limited to few architectures like ARM and MIPS
- static analysis is well covered by many (non-)commercial solutions
Because …

- it can be used to do remote testing with frame-conditions
- it was helpful for a lot of research/customer-projects
- dynamic analysis is currently limited to few architectures like ARM and MIPS
- static analysis is well covered by many (non-)commercial solutions
Nowadays Firmware Development

Bear in mind that nobody wants to do work twice! Buildroot is a very handy tool.
Bear in mind that nobody wants to do work twice! Buildroot is a very handy tool.

The most projects regarding development of embedded devices are based on one or multiple software/hardware development kits.
Nowadays Firmware Development

Bear in mind that nobody wants to do work twice! Buildroot is a very handy tool.

The most projects regarding development of embedded devices are based on one or multiple software/hardware development kits.

End-of-life, time-to-market and outsourcing are prominent buzzwords/-phrases.
Bear in mind that nobody wants to do work twice! Buildroot is a very handy tool.

The most projects regarding development of embedded devices are based on one or multiple software/hardware development kits.

End-of-life, time-to-market and outsourcing are prominent buzzwords/-phrases.

VENDOR CONTACT TIMELINE

2017-07-10: Contacting vendor through security@linksys.com. Set release date to 2017-08-29.
2017-07-12: Confirmation of recipient. The contact also states that the unit is older and they have to look for it.
2017-08-07: Asking for update; Contact responds that they have to look for such a unit in their inventory.
2017-08-08: Contact responds that he verified three of four vulnerabilities.
2017-08-09: Sent PCAP dump and more information about vulnerability #4 to assist the contact with verification.
2017-08-18: Sending new advisory version to contact and asking for an update; No answer.
2017-08-22: Asking for an update; Contact states that he is trying to get a fixed firmware from the OEM.
2017-08-24: Asked the vendor how much additional time he will need.
2017-08-25: Vendor states that it is difficult to get an update from the OEM due to the age of the product ("Many of the engineers who originally worked on this code base are no longer with the company").
Nowadays Firmware Development

Bear in mind that nobody wants to do work twice! Buildroot is a very handy tool.

The most projects regarding development of multiple software/hardware development kits

End-of-life, time-to-market and outsourcing are prominent buzzwords/-phrases.

VENDOR CONTACT TIMELINE

2017-07-10: Contacting vendor through security@linksys.com. Set release date to 2017-08-29.
2017-07-12: Confirmation of recipient. The contact also states that the unit is older and they have to check it.
2017-08-07: Asking for update; Contact responds that they have to look for such a unit in their inventory.
2017-08-08: Contact responds that he verified three of four vulnerabilities.
2017-08-09: Sent PCAP dump and more information about vulnerability #4 to assist the contact with verification.
2017-08-18: Sending new advisory version to contact and asking for an update; No answer.
2017-08-22: Asking for an update Contact states that he is trying to get a fixed firmware from the OEM.
2017-08-24: Asked the vendor how much additional time he will need.
2017-08-25: Vendor states that it is difficult to get an update from the OEM due to the age of the product.
("Many of the engineers who originally worked on this code base are no longer with the company").
Nowadays Firmware Development

Bear in mind that nobody wants to do work twice! Buildroot is a very handy tool.
The most projects regarding development of embedded devices are based on one or multiple software/hardware development kits.

End-of-life, time-to-market and outsourcing are prominent buzzwords/-phrases.

VENDOR CONTACT TIMELIN
2017-07-10: Contacting vendor through sales representative
2017-07-12: Confirmation of recipient. The contact is assigned to the right person for it.
2017-08-07: Asking for update; Contact responds that he will check and send.
2017-08-08: Contact responds that he verified the issues but no solution.
2017-08-09: Sent PCAP dump and more information to vendor for verification.
2017-08-18: Sending new advisory version.
2017-08-22: Asking for an update.
2017-08-24: Asked the vendor how much it would cost to develop a custom software.
2017-08-25: Vendor states that it is difficult to develop a custom software.

(Many of the engineers who originally worked on this code base are no longer with the company.)
Nowadays Firmware Development – Storage

In most cases, a small external flash chip or a SD-card is used.
Nowadays Firmware Development – Storage

In most cases, a small external flash chip or a SD-card is used. These chips are usually SPI, NAND or NOR flash memory ICs.
In most cases, a small external flash chip or a SD-card is used. These chips are usually SPI, NAND or NOR flash memory ICs. They are either directly flashed with a programmer or written via JTAG or another debug-interface.
In most cases, a small external flash chip or a SD-card is used. These chips are usually SPI, NAND or NOR flash memory ICs. They are either directly flashed with a programmer or written via JTAG or another debug-interface.

A part of this memory is also used to store configurations like usernames, PINs etc. which is called NVRAM (Non-Volatile RAM). It used to be a separate IC.
Distribution:

- Web-sites, FTP-servers, or as physical mail.
- Some vendors also use push-messages in the web-interface of the device.
Nowadays Firmware Development – Distribution / Device Upload

Distribution:
• Web-sites, FTP-servers, or as physical mail.
• Some vendors also use push-messages in the web-interface of the device.

Upload:
• Web-interface
• USB stick / SD-card
• TFTP / FTP
• Via an external programmer (JTAG)
Extracting Firmware can be done in by:

- JTAG/ISP/SWD programmers – this can be locked for some chips!
- Chip-off techniques – remove the flash chip and dump it directly. Have a look at our SEC Xtractor project (https://github.com/sec-consult/)
- Sniffing – can be done for a broad range of serial/parallel communication interfaces. Most prominent example is SPI.
- Side-Channel attacks – Glitching can lead to malfunctions for instructions. In specific cases, a UART interface can be abused to print out the whole content of the firmware.
- Microscopy – by using a SEM, an internal flash memory can be dumped in an optical way. Other ways are possible with microsurgery.
...and much more.
First of all, past publications about multi-arch firmware emulation were studied:

- Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces (Costin et al.) – standard Debian Images / chroot in the target firmware
- FIRMADYNE (Chen et al.) – modified kernels with musl-libc / target firmware file system is directly used

Both projects were mostly covering ARM(EL), MIPS(BE/EL)
First of all, past publications about multi-arch firmware emulation were studied:

- Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces (Costin et al.) – standard Debian Images / chroot in the target firmware
- FIRMADYNE (Chen et al.) – **modified kernels** with musl-libc / target firmware file system is directly used

Both projects were mostly covering ARM(EL), MIPS(BE/EL)

Combined with Buildroot and QEMU
Preparations

The following tasks were crucial to create an environment where the target firmware feels comfortable:

- Find out where the root file system is located → important for the chroot command
- Find out what architecture and instruction set is used → ARMv4 != ARMv7
- Find out which C library is used → uClibc, musl-libc or glibc
- Prepare a system startup script → inittab? rcS? Scripts in rc.d/ or init.d/?
The first thought I had was: “that must be solved with a complex algorithm”

But it was much simpler: Do it graphically with a folder keyword search for UNIX based systems.

The only constraint was, that it must be a Linux-based firmware with a file system.

Locate the Root File System

```
ubifs-root/207333087/ubifs/usr/sbin/sct_client
ubifs-root/207333087/ubifs/usr/sbin/wpa_supplicant
ubifs-root/207333087/ubifs/usr/sbin/get_devices_uuid
ubifs-root/207333087/ubifs/usr/sbin/ubiformat
ubifs-root/207333087/ubifs/usr/sbin/contrack_parse
ubifs-root/207333087/ubifs/usr/sbin/pub_autochannel_config
ubifs-root/207333087/ubifs/usr/sbin/htpasswd
ubifs-root/207333087/ubifs/usr/sbin/ehtpasswd
ubifs-root/207333087/ubifs/usr/sbin/speedtest_down
ubifs-root/207333087/ubifs/usr/sbin/iwconfig
ubifs-root/207333087/ubifs/usr/sbin/contrack
ubifs-root/207333087/ubifs/usr/sbin/rssi_to_rcpi
ubifs-root/207333087/ubifs/usr/sbin/update_device_db
ubifs-root/207333087/ubifs/usr/sbin/tess_steer_local_decision_eng
ubifs-root/207333087/ubifs/usr/sbin/radvd
ubifs-root/207333087/ubifs/usr/sbin/brctl
ubifs-root/207333087/ubifs/usr/sbin/porter
ubifs-root/207333087/ubifs/usr/sbin/bluetoothctl
ubifs-root/207333087/ubifs/usr/sbin/lbd
ubifs-root/207333087/ubifs/usr/sbin/cfg_restore
ubifs-root/207333087/ubifs/usr/sbin/acs
ubifs-root/207333087/ubifs/usr/sbin/pub_nb_rssi
ubifs-root/207333087/ubifs/usr/sbin/ipv4_firewall
ubifs-root/207333087/ubifs/usr/sbin/pub.pic_link_status_changed
```
Locate the Root File System

Using this kind of location, a histogram of multiple possible root file systems can also be created.

Do not rely on key binaries like busybox or bash as they can also be located in a rescue file system.

Use plausibility checks, like “are there even executables in the detected file system?”
Using this kind of location, a histogram of multiple possible root file systems can also be created. Do not rely on key binaries like `busybox` or `bash` as they can also be located in a rescue file system. Use plausibility checks, like "are there even executables in the detected file system?".

Locate the Root File System

```
/romfs_slave-x.squashfs.img.extracted/squashfs-root
/romfs_slave-x.squashfs.img.extracted/squashfs-root/mnt/mtd/Log
/romfs_slave-x.squashfs.img.extracted/squashfs-root/mnt/custom
/romfs_slave-x.squashfs.img.extracted/squashfs-root/mnt/mtd_backup
/romfs_slave-x.squashfs.img.extracted/squashfs-root/usr/bin/lua/com
/romfs_slave-x.squashfs.img.extracted/squashfs-root/usr/bin/ssl
/romfs_slave-x.squashfs.img.extracted/squashfs-root/mnt/mtd_backup
/romfs_slave-x.squashfs.img.extracted/squashfs-root/mnt/mtd_Log
/romfs_slave-x.squashfs.img.extracted/squashfs-root/mnt/mtd_backup
/romfs_slave-x.squashfs.img.extracted/squashfs-root/mnt/mtd_backup
```

Root File-System Detection
Identify the Architecture

A common way are the tools “readelf” and “file”. But for emulation, a more precise way to identify the exact instruction set is necessary. Other ways are:

- Looking for the “vermagic” string in kernel modules
- Looking for symbols that contain keywords like “ARM7TDMI” or “MIPS32R5”
- If everything fails, grep all executables to find the instruction set (bad success rate)
Identify the Architecture

A common way are the tools “readelf” and “file”. But for emulation, a more precise way to identify the exact instruction set is necessary. Other ways are:

- Looking for the “vermagic” string in kernel modules for “ARM7TDMI” or “MIPS32R5”
- Looking for symbols that contain keywords like “ARM7TDMI” or “MIPS32R5”
- If everything fails, grep all executables to find the instruction set (bad success rate)

The exact instruction set matters!
The interpreter for executables is especially relevant for cross-compiling binaries. They can be easily determined and constitute another important detail that must be considered!

This is crucial for ARM Hard-Float, as it has effect to the QEMU virtual machine.
Startup scripts in firmwares are placed on different locations on the system.

A straightforward way that was used to start the most firmwares of the sample set is parsing the `inittab` file.

A simple script was written that searches the `inittab` file in the extracted firmware image and also prints a graph of the different commands:

If no `inittab` file is present on the system, other typical startup pointers are `/etc/rcS`, `/etc/init.d/rcS` or `/etc/rc.d/rcS`.
Pre-Analysis – Sample Set of (Almost) 200 Firmwares across 49 Vendors

Interpreters that were found in all the firmwares (sometimes there is more than one):

<table>
<thead>
<tr>
<th>uClibc</th>
<th>libc (ld-linux/ld)</th>
<th>libc hard-float</th>
<th>musl libc</th>
<th>musl hard-float</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>103 (78/25)</td>
<td>11</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Architectural distribution:

<table>
<thead>
<tr>
<th>Architectural Distribution</th>
<th>ARMv8 (BE/EL)</th>
<th>ARMv7 (BE/EL)</th>
<th>ARMv6 (BE/EL)</th>
<th>ARMv5 (BE/EL)</th>
<th>ARMv4 (BE/EL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv8 (BE/EL)</td>
<td>0 / 2</td>
<td>0 / 58</td>
<td>0 / 3</td>
<td>0 / 34</td>
<td>1 / 5</td>
</tr>
<tr>
<td>MIPS32 (BE/EL)</td>
<td>39 / 36</td>
<td>3 / 0</td>
<td>10 / 0</td>
<td>4 / 1</td>
<td>1 / 2</td>
</tr>
</tbody>
</table>
Pre-Analysis – Sample Set of (Almost) 200 Firmwares across 49 Vendors

Interpreters that were found in all the firmwares (sometimes there is more than one):

<table>
<thead>
<tr>
<th></th>
<th>uClibc</th>
<th>libc (ld-linux/ld)</th>
<th>libc hard-float</th>
<th>musl libc</th>
<th>musl hard-float</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>103</td>
<td>103 (78/25)</td>
<td>11</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Architectural distribution: ARM is very often used as little endian architecture, but it differs for MIPS!

<table>
<thead>
<tr>
<th></th>
<th>ARMv8 (BE/EL)</th>
<th>ARMv7 (BE/EL)</th>
<th>ARMv6 (BE/EL)</th>
<th>ARMv5 (BE/EL)</th>
<th>ARMv4 (BE/EL)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 / 2</td>
<td>0 / 58</td>
<td>0 / 3</td>
<td>0 / 34</td>
<td>1 / 5</td>
</tr>
<tr>
<td>MIPS32 (BE/EL)</td>
<td>39 / 36</td>
<td>3 / 0</td>
<td>10 / 0</td>
<td>4 / 1</td>
<td>1 / 2</td>
</tr>
</tbody>
</table>
Pre-Analysis – Sample Set of (Almost) 200 Firmwares across 49 Vendors

Interpreters that were found in all the firmwares (sometimes there is more than one):

<table>
<thead>
<tr>
<th>uClibc</th>
<th>libc (ld-linux/ld)</th>
<th>libc hard-float</th>
<th>musl libc</th>
<th>musl hard-float</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>103 (78/25)</td>
<td>11</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Architectural distribution:

<table>
<thead>
<tr>
<th>ARMv8 (BE/EL)</th>
<th>ARMv7 (BE/EL)</th>
<th>ARMv6 (BE/EL)</th>
<th>ARMv5 (BE/EL)</th>
<th>ARMv4 (BE/EL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 / 2</td>
<td>0 / 58</td>
<td>0 / 3</td>
<td>0 / 34</td>
<td>1 / 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIPS32 (BE/EL)</th>
<th>MIPS64 (BE/EL)</th>
<th>PowerPC (BE/EL)</th>
<th>X86 / X86_64</th>
<th>AVR / ARC5</th>
</tr>
</thead>
<tbody>
<tr>
<td>39 / 36</td>
<td>3 / 0</td>
<td>10 / 0</td>
<td>4 / 1</td>
<td>1 / 2</td>
</tr>
</tbody>
</table>

Cavium Octeon: Documentation available at
http://vsevteme.ru/network/169/attachments/show?content=297548
http://vsevteme.ru/network/169/attachments/show?content=297550

Very specific architectures
E.g. SH4 is used for other industries
Our workflow:

1) Pre-analysis of the target firmware
2) Creating a suitable firmware image with kernel and userland (for analysis)
3) Copy the identified root file-system into the created firmware image
4) Start the firmware image and use chroot to switch into the target firmware
5) Run all startup scripts
6) Security analysis

Done!
Our workflow:

1) Pre-analysis of the target firmware

2) Creating a suitable firmware image with kernel and userland (for analysis)

3) Copy the identified root file-system into the created firmware image

4) Start the firmware image and use `chroot` to switch into the target firmware

5) Run all startup scripts

6) Security analysis

A cross-compiler can also be generated!
Our workflow:

1) Pre-analysis of the target firmware
2) Creating a suitable firmware image with kernel and userland
3) Copy the identified root file-system into the created firmware image
4) Start the firmware image and use `chroot` to switch into the target firmware
5) Run all startup scripts
6) Security analysis

A cross-compiler can also be generated!
Preparring Fake Images – Buildroot to the Rescue!

Our workflow:

1) Pre-analysis of the target firmware
2) Creating a suitable firmware image with kernel and userland (for analysis)
3) Copy the identified root file-system into the created firmware image
4) Start the firmware image and use chroot to switch into the target firmware
5) Run all startup scripts
6) Security analysis
Preparing Fake Images – Buildroot to the Rescue!

Our workflow:

1) Pre-analysis of the target firmware
2) Creating a suitable firmware image with kernel and userland (for analysis)
3) Copy the identified root file-system into the created firmware image
4) Start the firmware image and use chroot to switch into the target firmware
   **5) Run all startup scripts**
6) Security analysis

Does not work for all cases.
Sometimes just partial images are available!
Our workflow:

1) Pre-analysis of the target firmware
2) Creating a suitable firmware image with kernel and userland (for analysis)
3) Copy the identified root file-system into the created firmware image
4) Start the firmware image and use chroot to switch into the target firmware
5) Run all startup scripts
6) Security analysis

Does not work for all cases. Sometimes just partial images are available!
A lot of architectures were covered for this project.

The target firmware is embedded into the firmware image to keep the network traffic low. Mounting via NFS was the first way how it was done, but that turned out to be not optimal for monitoring and debugging.

A bridged network was used within QEMU to start the firmware with a dedicated PC.

By changing the hardware address for each firmware with QEMU command line parameters and using DHCP, the full process can be designed scalable.

Loading kernel modules is only possible when the version is fitting!
Firmware Emulation Demo – Runtime!
Monitoring and Debugging

To watch the firmware startup and called commands and network daemons, monitoring and debugging is an important step. Findings for an easier life:

- Most important are the Linux commands `ps`, `top`, `netstat` and `tcpdump`.
- During the evaluation, a good portion of valuable information was gathered just by dumping the output of `netstat` and `ps`.
- The painful cross-compilation can be skipped as Buildroot covers this :)
- Static builds of `gdb(-server)`, `strace`, `ltrace`, `valgrind` and other tools can be done with this toolchain.

The kernel can also be customized with Buildroot:

- enable tracing and use `perf` to get all calls!
- this can be done with `$ make linux-menuconfig`
Study Samples from ...

[Logos of various companies]
Study Outcome of Linux Based Firmware Emulation

Emulation success rate (tested with sh/ash/bash) 178/199 (~89%)
One or more inferred TCP listeners 31/199 (~16%)
One or more inferred UDP listeners 15/199 (~8%)
A lot firmware images were incomplete which is the reason why many services could not have been started. This does not mean that the emulation itself wasn't successful!

Known vulnerabilities that were tested automatically:

- CVE-2015-7547 (glibc getaddinfo buffer overflow) 8/199 (~4%)
- CVE-2015-0235 (GHOST buffer overflow) 28/199 (~14%)
- Shellshock (multiple CVEs) 1/199 (~0.5%)
Published:

- Command Injection in Phoenix Contact Devices
- Hardcoded Credentials & Vulnerable TPS in Cisco SMB Routers
- Hardcoded Key Material & Vulnerable TPS in WAGO Managed Industrial Switches

Unpublished but already communicated within our responsible disclosure process to the vendor:

- Multiple Vulnerabilities in one Red Lion Device
- Multiple Vulnerabilities in Korenix Devices

More vulnerabilities that must be communicated...
By loading the “cfg” CGI binary into Ghidra, the vulnerable code can be spotted very fast:

```c
140  uVar4 = scan_boundary(0.*(undefined4 *)(param_2 + 8)).3.0.0;
141  return uVar4;
142
143  if (param_3 != 0) {
144  
145  }
146  if (*(int *)(param_1 + 0x10) != 1) goto LAB_000003e8;
147  if (*(char **)(param_1 + 0x14) == *(char *)0x0) {
148  html_print(1,"<pre>$s</pre>\r\n", "missing filename");
149  goto LAB_000003e8;
150  }
151  sl = strchr(*(char **)(param_1 + 0x14), 0x2e);
152  if (!sl == *(char *)0x0) {
153  uVar4 = *(undefined4 *)(param_1 + 0xc);
154  LAB_000004bc:
155  sl = ";
156  }
157  else {
158  sl = sl + 1;
159  uVar4 = *(undefined4 *)(param_1 + 0xc);
160  if (_sl == *(char *)0x0) goto LAB_000004bc;
161  }
162  run_shell(0x1000,"/usr/sbin/import_cfg /tmp/cfg_import %s/new_config %s", uVar4, __sl);
163  html_print(1,"<pre>$s</pre>\r\n", "please reboot next");
164  LAB_000003e8:
165  print_foot();
166  free(*(void **)(param_1 + 0x14));
167  remove_dir_leaf(*(undefined4 *)(param_1 + 0xc));
168  return 0xfffff00f;
169  }
```
Emulating firmware with QEMU and Buildroot while covering different architectures works really good!

Tested approaches were:

- Pre-built Debian images → no kernel modifications possible, changes in the userland are hard.
- Building the kernel from scratch → kernel modifications are really complex, only good when you are familiar with Linux kernel internals.
- Using the target firmware’s file system only → observation must be done via QEMU and the kernel, manual testing is hard.

Improvements:

- Implement Cavium Octeon to QEMU (KVM already supports this architecture)
- Use kernel hopping in Buildroot → enables loading of some kernel modules
- Library resolving, e.g. by using scanelf → helps to reconstruct the file system
- Pre-emulation with QEMU user mode → better architecture detection
Thank You!

THAT'S IT

QUESTIONS?