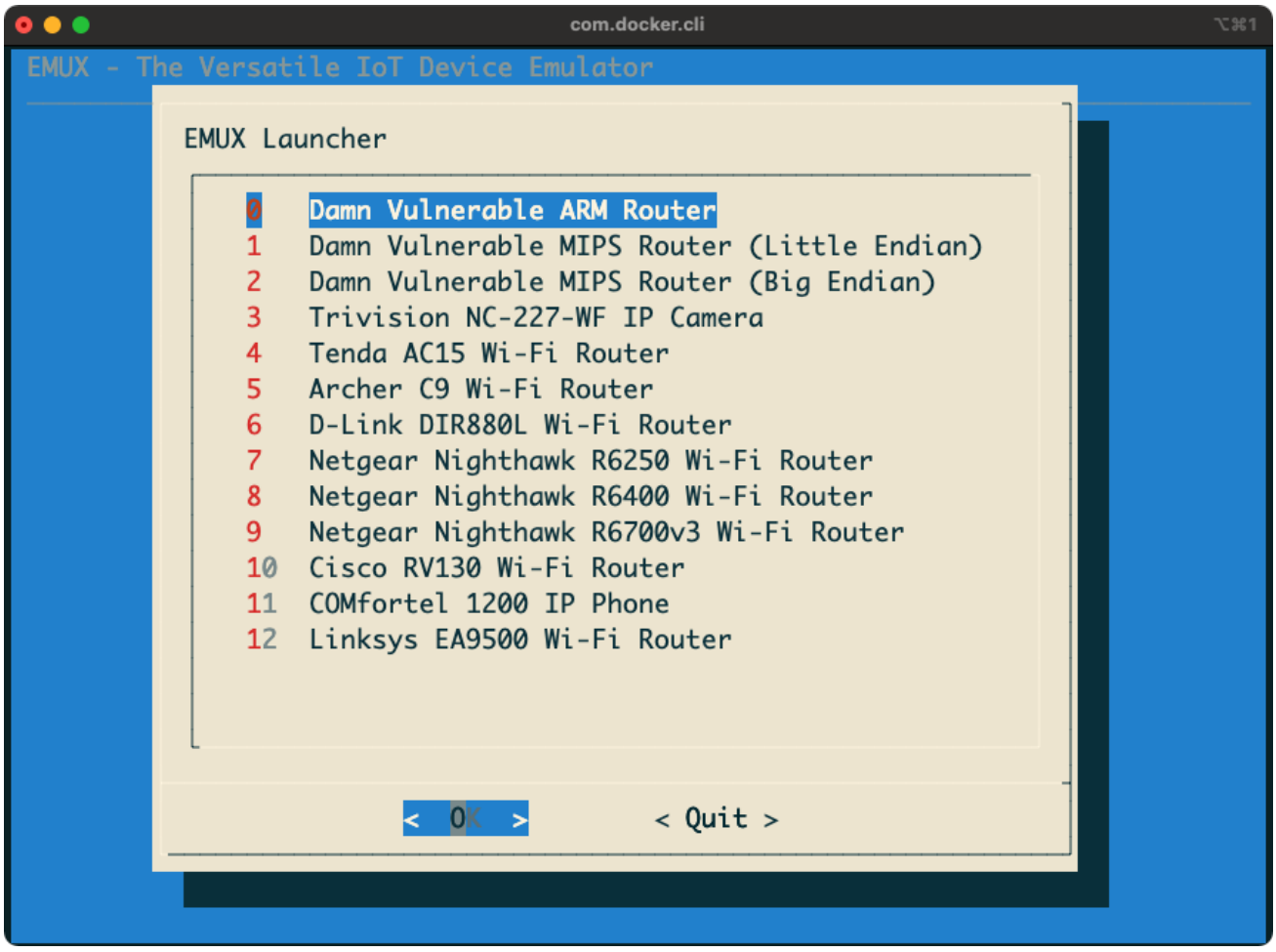
EMUX (formerly ARMX) Firmware Emulation Framework

by Saumil Shah @therealsaumil

April 2022

Welcome, MIPS!

With the addition of MIPS, ARMX has changed its name to EMUX! Try out the **Damn Vulnerable MIPS Router** exercises included with the new EMUX Docker image.



A brand new Docker container running EMUX. Going ahead, all official EMUX releases shall be released as Docker images. Lightweight, Compact, Easy.

> ### Shut up and give me the g00diez
>
> Github: https://github.com/therealsaumil/emux
>
> A brand new EMUX Docker image is ready for use! The old "Preview VM" is now discontinued in favour of the Docker image.

QUICK INSTALL STEPS

Step 0 - Ensure that Docker is installed and running!

Test if Docker is working by running `hello-world`

```
docker run hello-world
```

Note: Ubuntu (and other Linux distros) users, ensure that your current user has privileges to run Docker as an administrator:

```
sudo groupadd docker
sudo gpasswd -a $USER docker
sudo usermod -aG docker $USER
```

Step 1 - Clone the EMUX repository

```
git clone --depth 1 --single-branch  https://github.com/therealsaumil/emux.git
```

Step 2 - Build the EMUX docker volume and image

```
cd emux
./build-emux-volume
./build-emux-docker
```

Note: If `build-emux-docker` fails, try and run it again by disabling `DOCKER_BUILDKIT`

```
- DOCKER_BUILDKIT=1 docker build -t $OWNERNAME/$IMAGENAME:$TAGNAME \
                                 -f Dockerfile-emux .
+ DOCKER_BUILDKIT=0 docker build -t $OWNERNAME/$IMAGENAME:$TAGNAME \
                                 -f Dockerfile-emux .
```

Step 3 - Run EMUX!

Open a terminal, and start the `emux-docker` container:

```
./run-emux-docker
```

You will be greeted with a purple shell prompt `[EMUX-DOCKER`  `:~$]` . After a while, it is common to have many terminals attached to the container. Coloured shell prompts makes it easy to remember where you are.

Next, start the EMUX `launcher` :

```
[EMUX-DOCKER 💪:~$] launcher
```

and select any emulated device that you wish to run.

Step 4 - Launch the emulated device's userland processes.

Next, open a new terminal and attach to the running `emux-docker` container:

```
./emux-docker-shell
```

All attached container shells have a blue shell prompt. Invoke the `userspace` command to bring up the userland processes of the emulated target:

```
[emux-docker shell 👶:~$] userspace
```

Read the documentation for more details.

INTRODUCING

The EMUX Firmware Emulation Framework is a collection of scripts, kernels and filesystems to be used with QEMU to emulate ARM and MIPS Linux IoT devices. EMUX is aimed to facilitate IoT research by virtualising as much of the physical device as possible. It is the closest we can get to an actual IoT VM.

Devices successfully emulated with EMUX so far:

- Damn Vulnerable ARM Router
- **Damn Vulnerable MIPS Router (Little Endian) [NEW!]**
- **Damn Vulnerable MIPS Router (Big Endian) [NEW!]**
- Trivision NC227WF Wireless IP Camera

- Tenda AC15 Wi-Fi Router (Github Docs)
- Archer C9 Wi-Fi Router

The following devices are not included with the public release, however they have been successfully emulated and used in training:

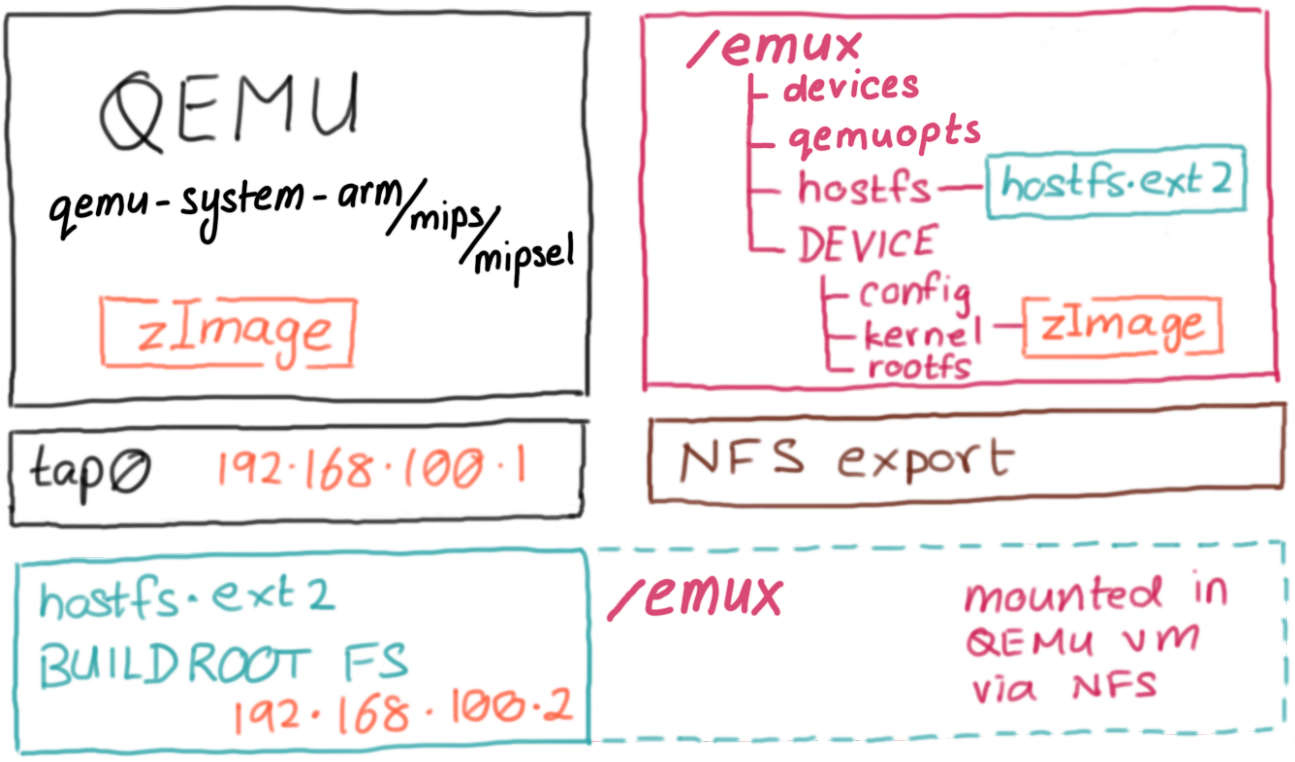
- D-Link DIR-880L Wi-Fi Router
- Netgear Nighthawk R6250 Wi-Fi Router
- Netgear Nighthawk R6400 Wi-Fi Router
- NEW! Netgear Nighthawk R6700v3 Wi-Fi Router
- Cisco RV130 Wi-Fi Router
- COMfortel 1200 VoIP Phone
- Linksys EA9500 Wi-Fi Router

Precursors of EMUX have been used in Saumil Shah's popular ARM IoT Exploit Laboratory training classes where students have found ~~four~~ several 0-day vulnerabilities in various ARM/Linux IoT devices.

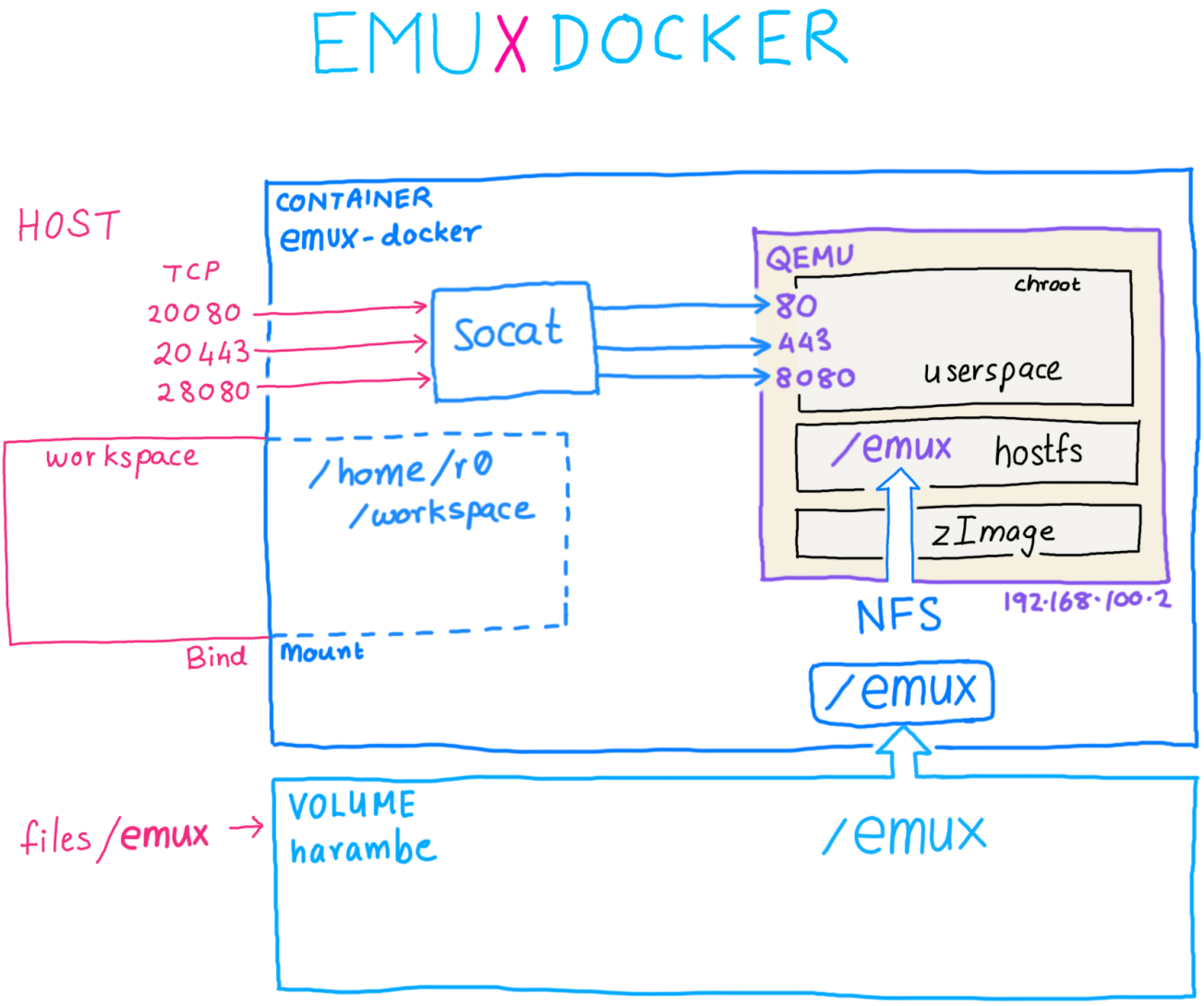
EMUX Architecture

EMUX is a collection of scripts, kernels and filesystems residing in the `/emux` directory. It uses `qemu-system-arm` , `qemu-system-mips` and `qemu-system-mipsel` to boot up virtual ARM and MIPS Linux environments. The `/emux` directory is exported over NFS to also make the contents available within the QEMU guest.

The host system running `qemu-system-arm|mips|mipsel` is assigned the IP address `192.168.100.1` and the QEMU guest is assigned `192.168.100.2` via `tap0` interface.



EMUX is packaged as a Docker image. The diagram below shows how the docker container is organised:

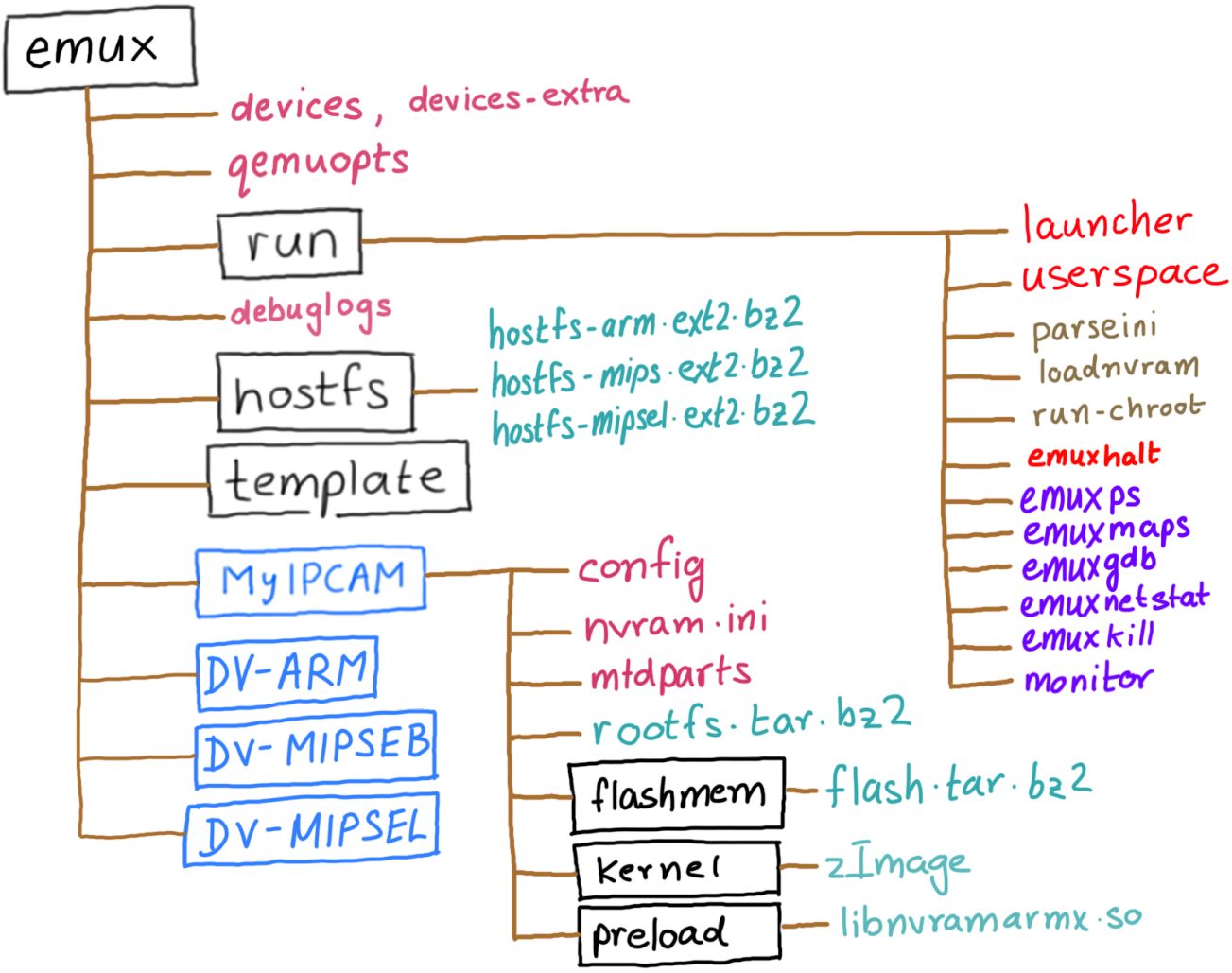


The docker image consists of:

- Volume `harambe` containing the `/emux` directory tree. (🦍 Harambe be praised!)
- Container `emux-docker` .
- Directory `workspace` on the host bind mounted as `/home/r0/workspace` in the container, to share files.
- NFS server running inside the container serving the `/emux` directory tree to emulated images running under QEMU
- Port forwarding from the host to QEMU running inside the container is done using `socat` .

The /emux directory

The `/emux` directory is organised as follows:



- `devices` : This file contains device definitions, one per line.
- `devices-extra` : Contains additional emulated devices not included in the general release. It is recommended that you add your own emulated devices to `devices-extra` .
- `qemuopts` : Abstracted QEMU options definitions for various types of QEMU Machines.

- `run/` : This folder contains scripts necessary to parse the device configuration, preload nvram contents and eventually invoke the userland processes of the device being emulated.
- `run/launcher` : The main script. `launcher` parses the `devices` file and displays a menu of registered devices. Selecting one of the devices will in turn invoke `qemu-system-arm` with the pre-defined QEMU options, corresponding Linux kernel and extracted root file system registered with the device.
- `run/userspace` : Start the userspace processes of an emulated device, once the kernel is booted up from the `launcher` .
- `debuglogs` : If present, it indicates the location where EMUX debugging logs will be written to. Extremely helpful in troubleshooting while creating a new emulated device.
- `template/` : Sample configuration and layout for a new device. Make a copy of the template when beginning to emulate a new IoT device.

The `run/` directory also contains a few commands that can be used from the host to interact with processes running within an EMUX emulated device.

- `emuxhalt` : Cleanly shut down the emulated device, and unmount all NFS mounts. Without a clean shutdown, there's always the risk of stale NFS handles.
- `emuxps` : Remotely enumerate processes running within EMUX.
- `emuxmaps` : Remotely dump the process memory layout of a process running within EMUX.
- `emuxnetstat` : Enumerate network sockets within EMUX.
- `emuxkill` : Remotely terminate a process running within EMUX.
- `emuxgdb` : Attach `gdb` to a process running within EMUX.
- `monitor` : Attach to the QEMU monitor.

`emuxps` , `emuxmaps` and `emuxgdb` are explained in detail in the Debugging With EMUX tutorial.

Contents of an emulated device

Each emulated device contains the following files/directories:

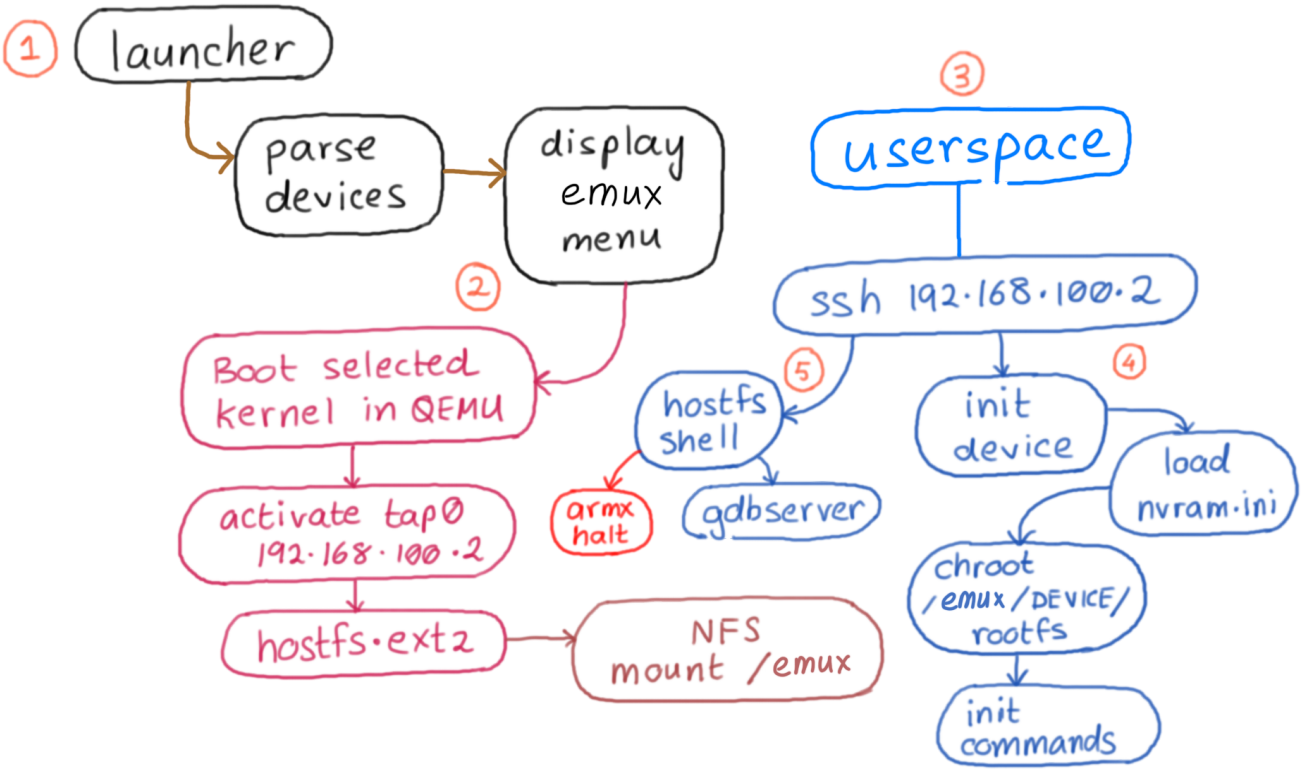
- `config` : Contains the device's name and description, ASLR settings, location of its root file system and commands to issue after the kernel has booted up and transferred control to the userland.
- `nvram.ini` : Contents of the device's non volatile memory, used for storing configuration settings. Contents of `nvram.ini` are preloaded into the emulated nvram before invoking the userland init scripts.
- `kernel/` : Contains a Linux kernel compiled (mostly via Buildroot) to closely match the properties of the emulated device such as kernel version, CPU support, VM_SPLIT,

supported peripherals, etc.

- `rootfs.tar.bz2` : A compressed archive containing the Root File System extracted from the target device. The name `rootfs.tar.bz2` is configurable from within the `config` file. EMUX will automatically unpack the Root File System the first time it is invoked.
- `flashmem/flash.tar.bz2` : A compressed archive containing two 64MB memory dump files `flash0.bin` and `flash1.bin` . These will be visible as a unified 128MB MTD Flash device.

Running an emulated device in EMUX

The diagram below describes each stage of EMUX:



There are five steps in running an emulated device:

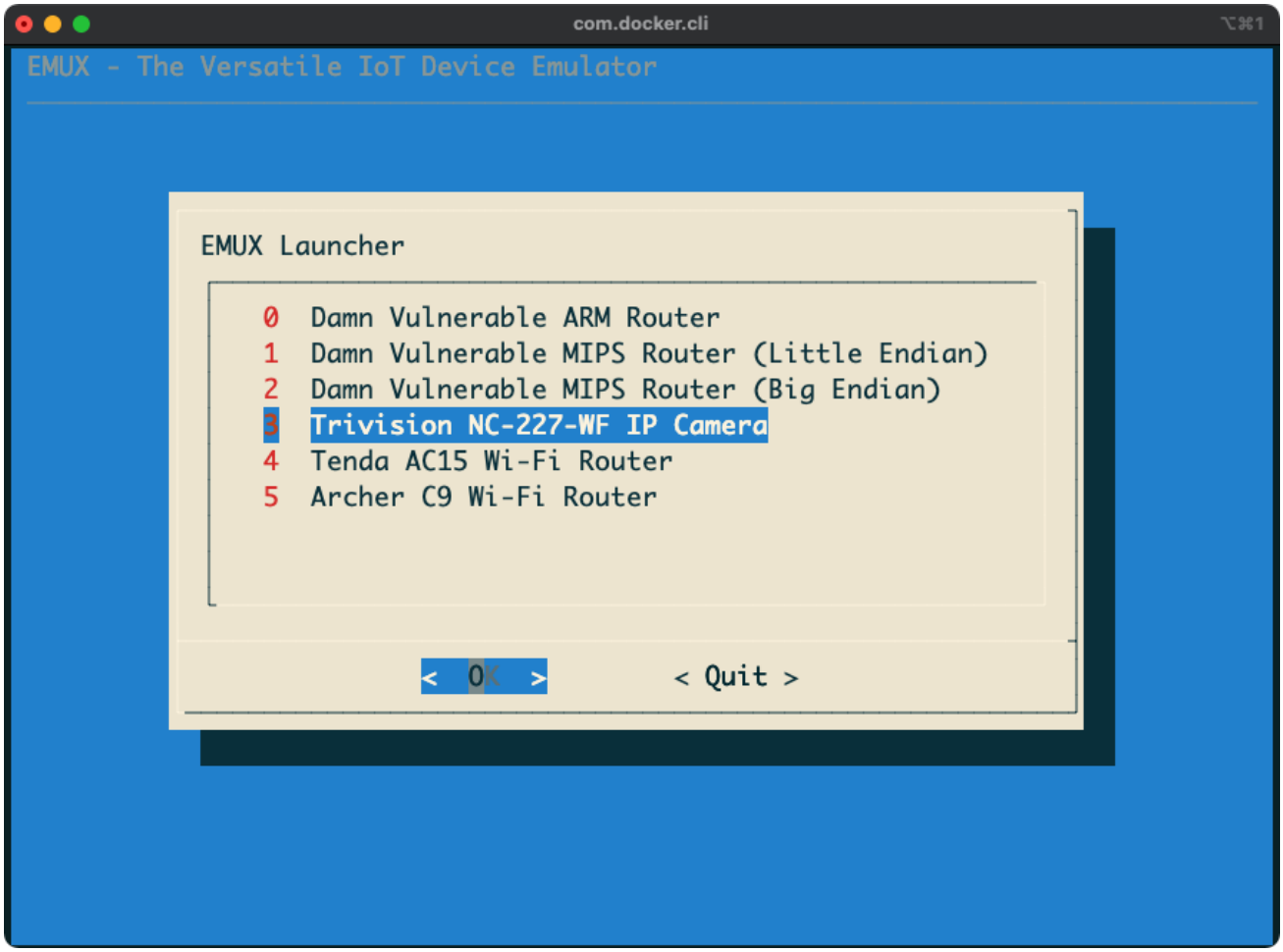
1. Launcher - choose from a list of available emulated devices
2. Select a device and boot its kernel and its hostfs
3. Userspace - choose from a list of available userspace actions
4. Start the devices' userspace processes
5. Optionally drop into the hostfs shell

Step 1: The Launcher

Invoke `launcher` .

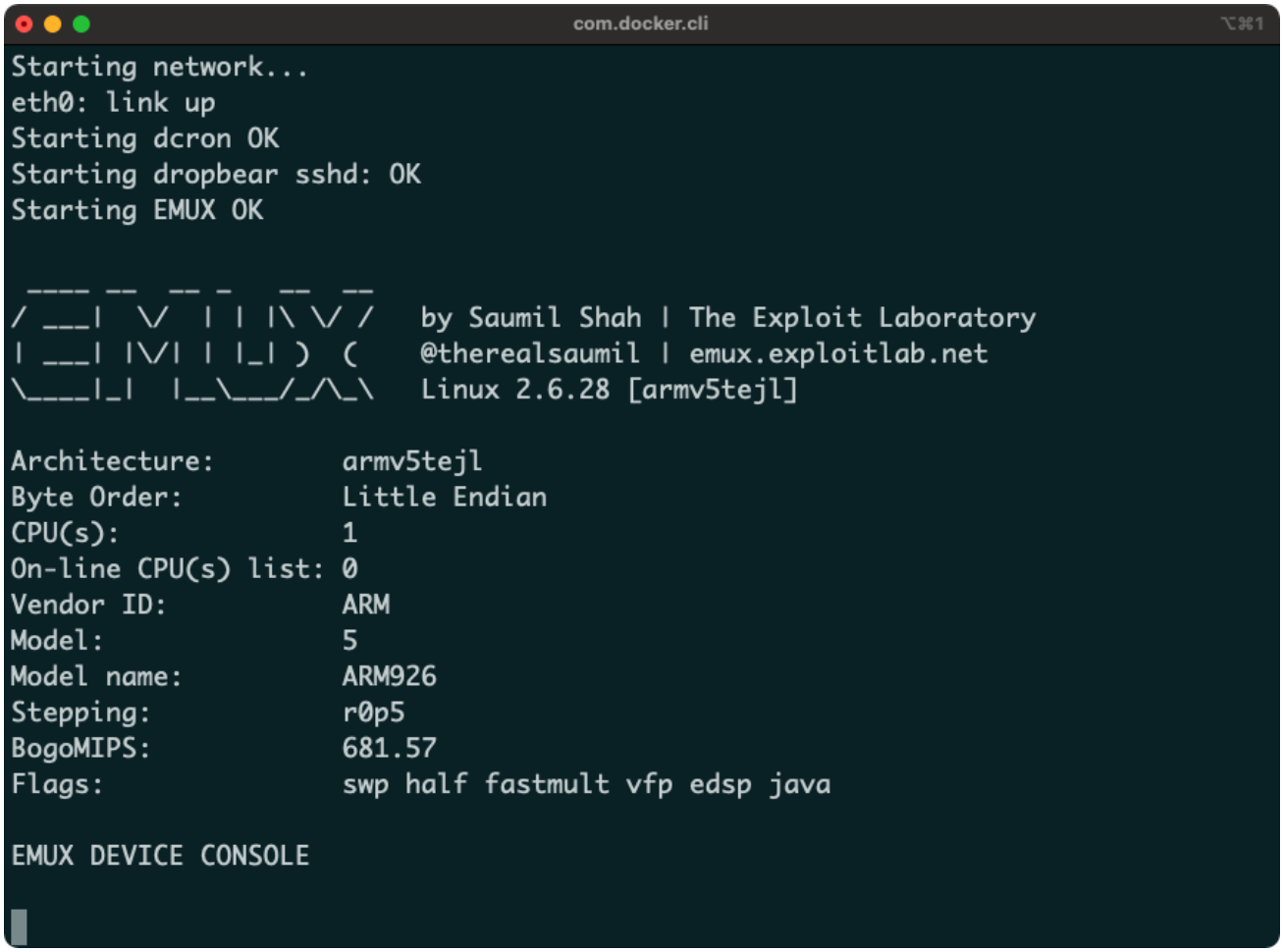
```
[+] Setting up forwarded ports 20080:80,20443:443,28080:8080
[+] mapping port 20080 -> 192.168.100.2:80
[+] mapping port 20443 -> 192.168.100.2:443
[+] mapping port 28080 -> 192.168.100.2:8080
  ___ __  __ _   __  __
 / __|  \/  | | |\ \/ /   by Saumil Shah | The Exploit Laboratory
 | __| |\/| | |_| )  (    @therealsaumil | emux.exploitlab.net
 \___|_|  |__\___/_/\_\

[EMUX-DOCKER 🐳] ~$ launcher
```

This will display a menu as shown below. In this example, we select the Trivision TRI227WF Wireless IP Camera.



Step 2: Start a device

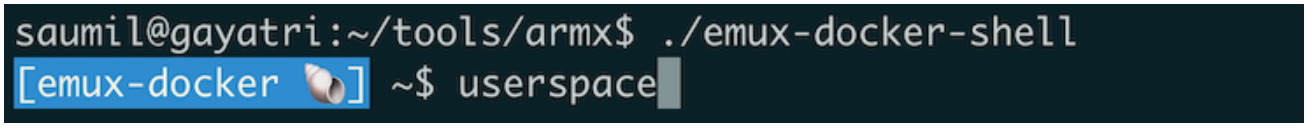
Selecting one of the devices will launch it under QEMU. The kernel which is included in the `kernel/` directory of the Trivision IP Camera's device configuration, is booted in `qemu-system-arm` and uses a pre-built Buildroot filesystem, which is referred to as `hostfs.ext2`. Host and guest IP addresses are assigned to `192.168.100.1` and `192.168.100.2` respectively.



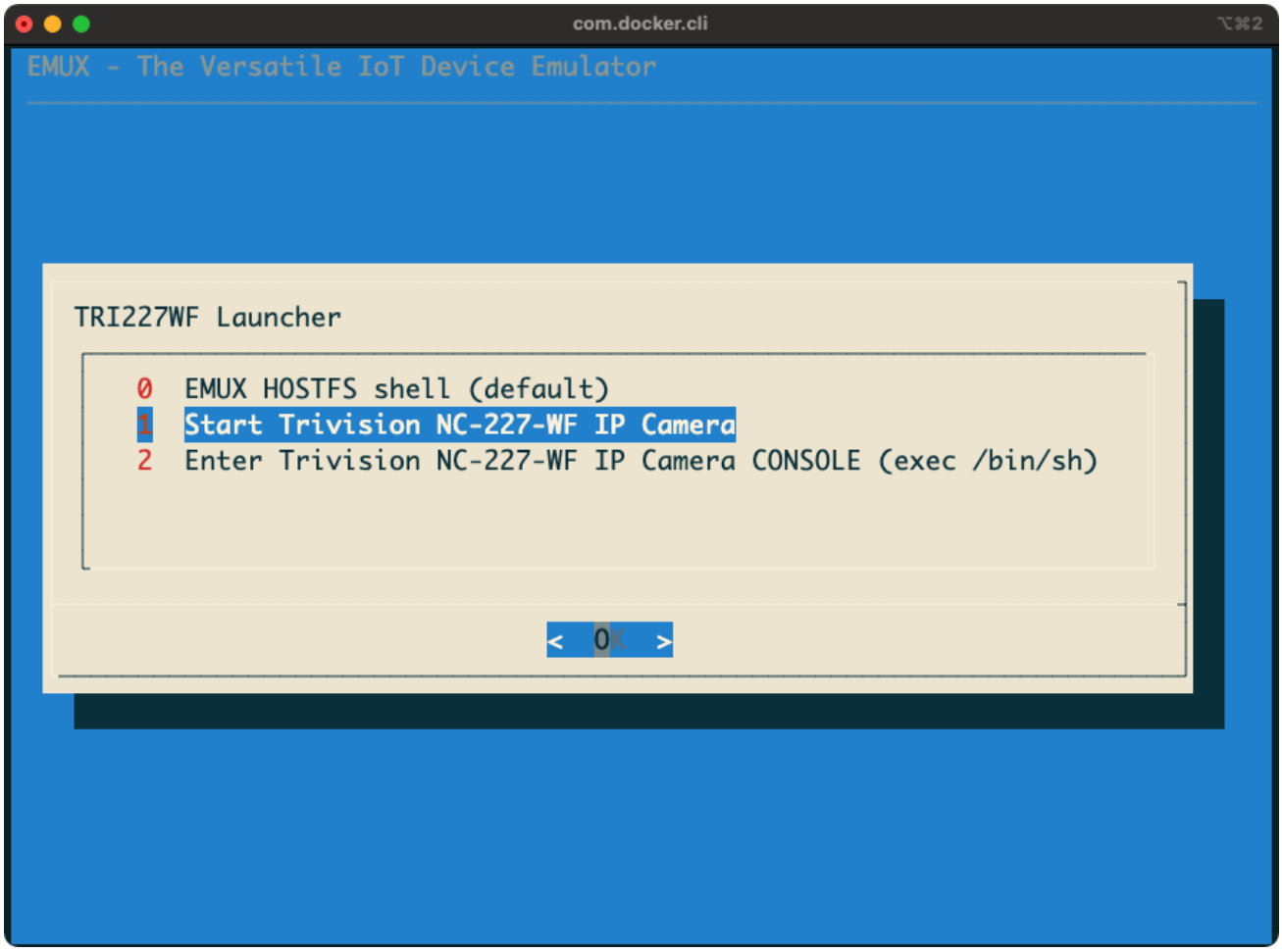
`hostfs-arm.ext2` , `hostfs-mips.ext2` and `hostfs-mipsel.ext2` contain several scripts and tools useful for running and dynamic analysis of the emulated device. The init scripts in `hostfs` mount the `/emux` directory over NFS. Thus, the contents of `/emux` are shared by both the host and the QEMU guest.

Step 3: Userspace

You will need to attach to the running `emux-docker` container and invoke the `userspace` command at the shell prompt.

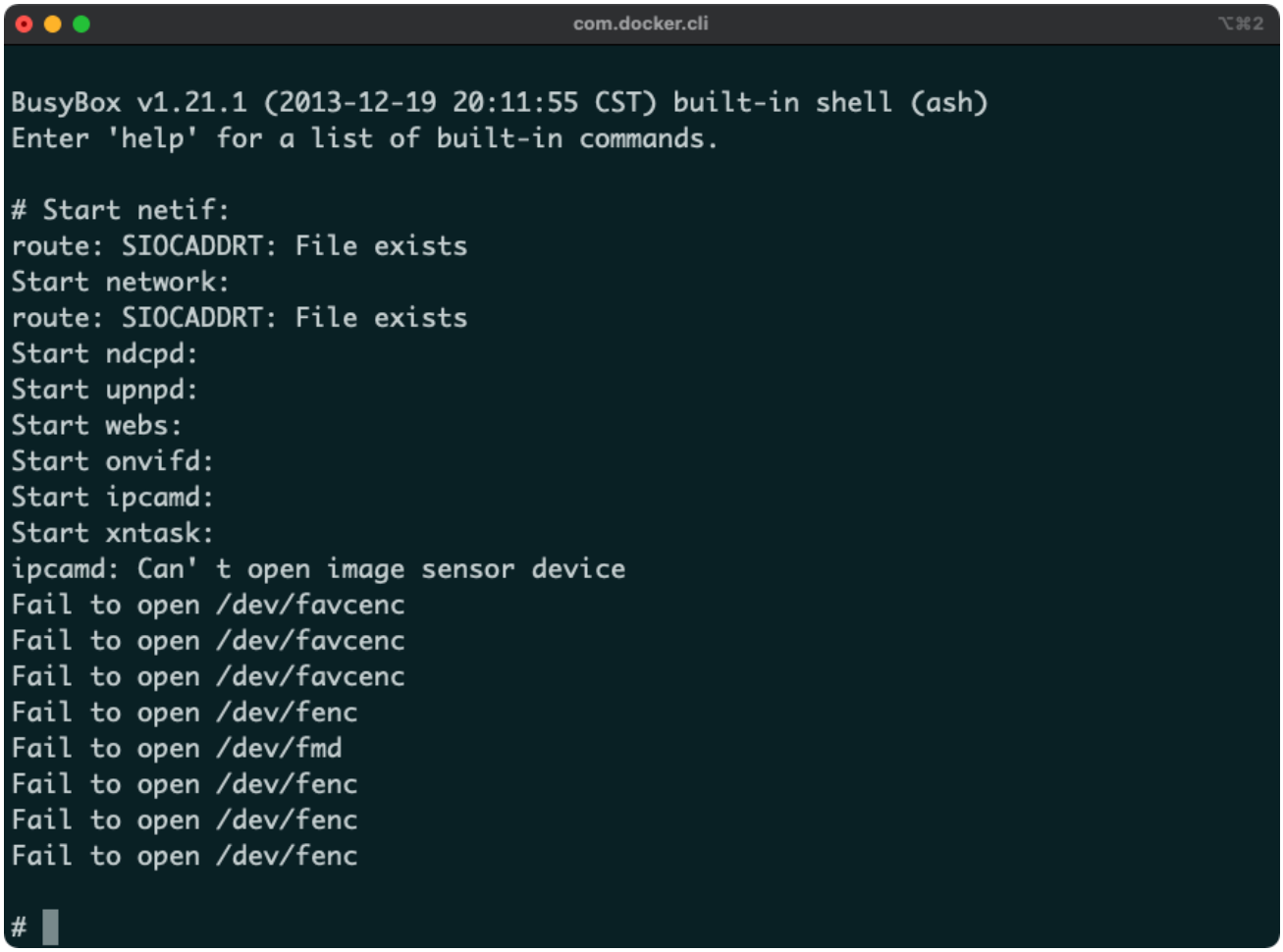


Internally the `userspace` command simply connects to the QEMU guest using SSH `ssh root@192.168.100.2` . This brings up a menu as shown below:



Step 4: Start the userspace processes

Selecting the option to launch the userspace processes of the device results in `run-init` being invoked from the corresponding device configuration directory within `/emux`. First, the contents of `nvram.ini` are loaded into the kernel's emulated nvram driver. Next, a `chroot` jail is created using the `rootfs` of the device. Lastly, the registered initialisation commands are invoked in the newly `chroot` ed `rootfs`, bringing up the device's services and init scripts.

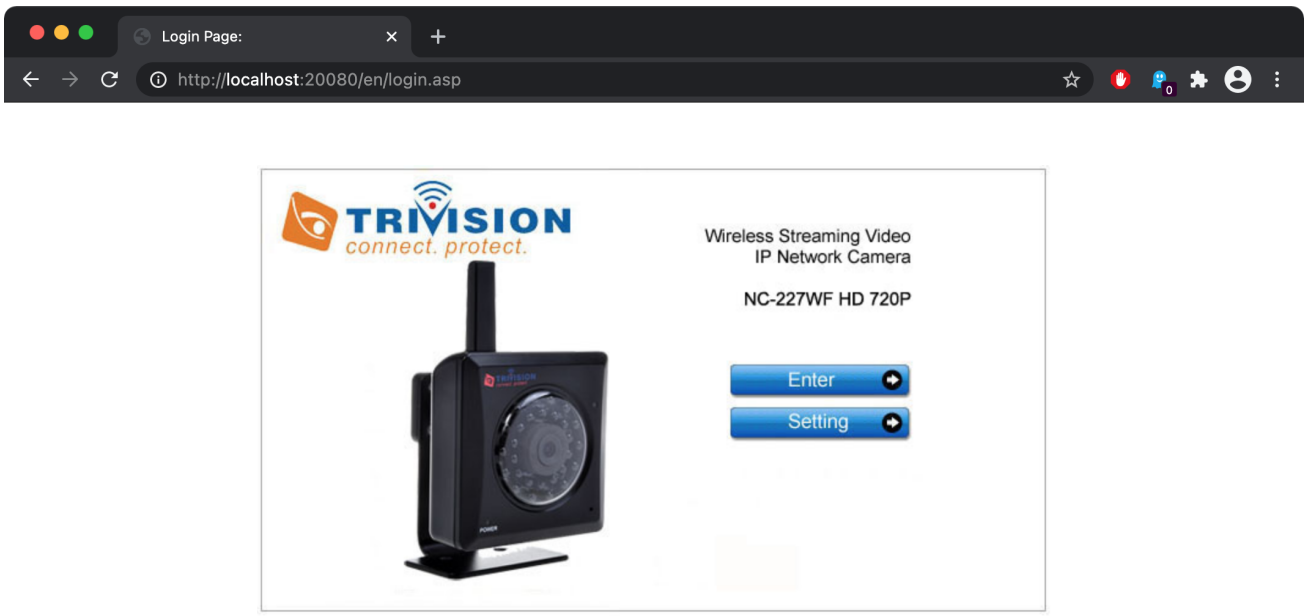


Step 5: Device booted up and ready

Once the device has fully "booted up" in EMUX, it is available for testing and analysis. The image below shows the administration interface of the IP Camera loaded in a browser. Note, to access the internal ports on `192.168.100.2` we will rely on port forwarding performed by `socat`. By default, the following ports are forwarded:

```
localhost:20080 -> 192.168.100.2:80
localhost:20443 -> 192.168.100.2:443
localhost:28080 -> 192.168.100.2:8080
```

To access the web administration interface for the booted up device, open a browser and navigate to `localhost:28000`. This in turn will forward your request to `192.168.100.2:80` inside the `emux-docker` container.



Overriding the forwarded ports

EMUX port forwarding is controlled by the `PORTFWD` environment variable. It is a comma separated list containing `FORWARDED_PORT:INTERNAL_PORT` pairs. To override the default port forwarding, simply set the contents of `PORTFWD` before invoking `run-emux-docker`:

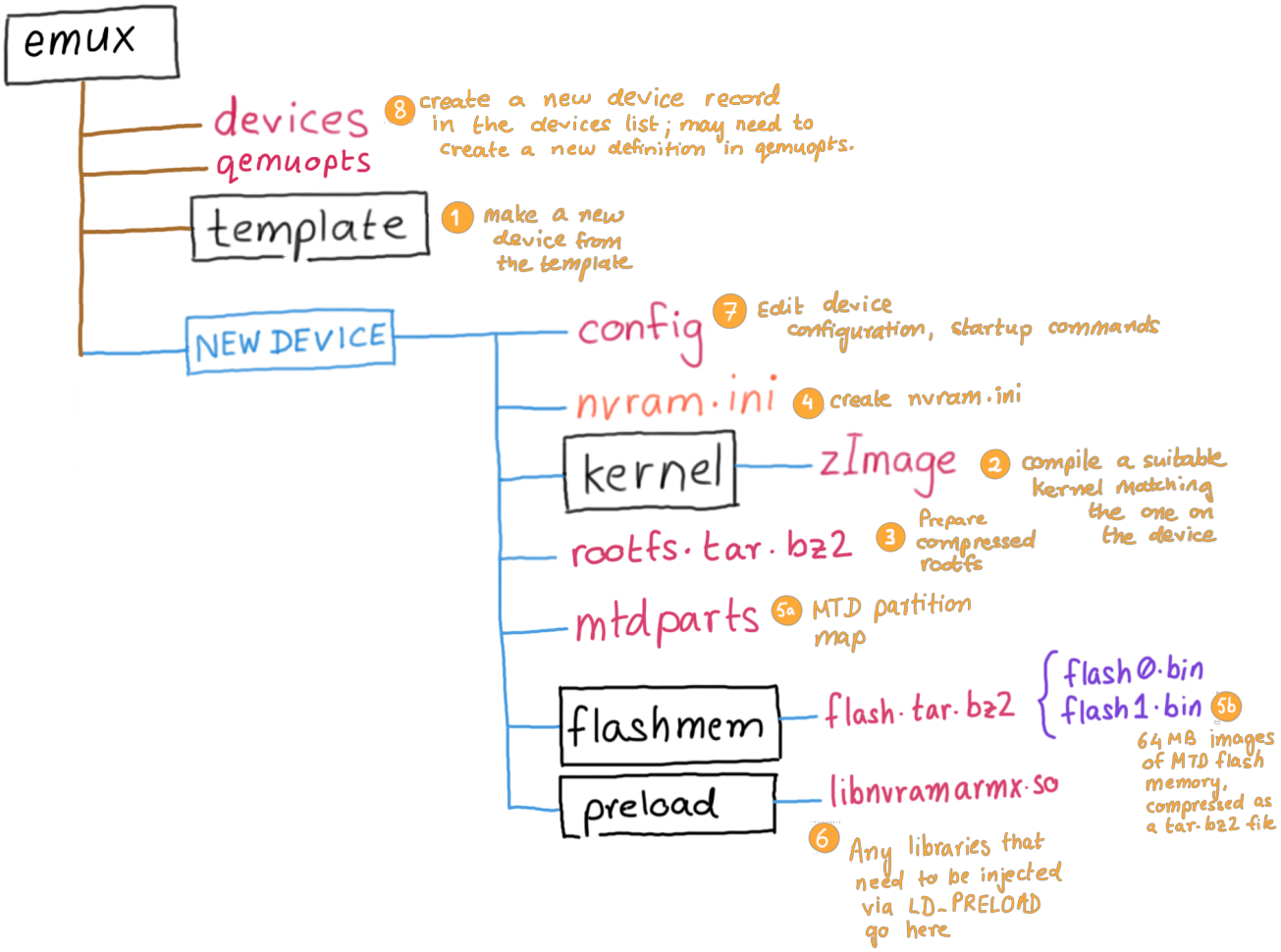
```
export PORTFWD="28000:8000,25800:5800"
./run-emux-docker
```

Creating your own emulated IoT Device

Before you begin to emulate an IoT device, you will need the following:

- Detailed analysis of the IoT device
- CPU (ARMv5/ARMv6/ARMv7/MIPS)
- Linux Kernel version
- Contents of the extracted flash memory (optional)
- Extracted Root File System from the flash memory
- Contents of nvram (optional)
- Generate a compatible kernel using Buildroot or Linux Kernel sources
- A week for troubleshooting!

The following diagram outlines the overall process of IoT device emulation.



Steps involved:

1. Copy the `template` directory to make a new device configuration.
2. Compile a matching kernel from source, and place it in the `kernel/` directory. You may also symlink an existing kernel if you wish to.
3. Copy the extracted `rootfs` from the device's firmware into the `rootfs/` directory. Typically these would be SquashFS or CramFS filesystems, uncompressed using `binwalk` or `unsquashfs` or `cramfsck`. Optionally you may also create a compressed `tar.bz2` archive of the root file system.
4. Place the contents of extracted nvram in `nvram.ini`
5. If you wish to emulate MTD flash, dump the contents of your device's flash memory and create two 64MB files named `flash0.bin` and `flash1.bin` and place them in the `flashmem/` directory. Optionally you may also compress them in a `tar.bz2` archive. You will then need to define the MTD partition layout to be passed to the kernel in the `mtdparts` file.
6. Place any shared libraries that you wish to inject using `LD_PRELOAD` in the `preload/` directory. Usually these shared libraries contain hooked functions necessary for certain emulated binaries to work properly.
7. Edit the `config` file with the newly populated device firmware contents.
8. Create a new device record in the `devices-extra` file. Pay close attention to QEMU command line options.

The following sample kernels are provided with the template.

- `zImage-2.6.39.4-vexpress` ARMv7 CPU on a `vexpress-a9` board.
- `zImage-2.6.31.14-realview-rv130-nothumb` ARMv6 CPU on a `realview-eb` board.
- `zImage-2.6.31-versatile-nothumb` ARMv5 CPU on a `versatilepb` board.
- `zImage-2.6.29.6-versatile` ARMv5 CPU on a `versatilepb` board.
- `zImage-2.6.28-versatile-nothumb` ARMv5 CPU on a `versatilepb` board.
- `vmlinux-3.18.109-malta-be` MIPS32 CPU (big endian) on a `malta` board. [NEW!]
- `vmlinux-3.18.109-malta-le` MIPS32 CPU (little endian) on a `malta` board. [NEW!]

However, it is encouraged to build a compatible kernel from source.

The EMUX Activity Log File

The June 2021 release of EMUX comes with a feature to enable activity logs. This comes in very handy in troubleshooting errors when adding a new device to EMUX. To enable logging, edit the `/emux/debuglogs` file:

```
# Uncomment logpath= to enable EMUX and QEMU console output logging.
# Only one logpath= should be uncommented.
#
logpath=/home/r0/workspace/logs/
#logpath=/emux/logs/
```

It is recommended to use `/home/r0/workspace/logs` since the `workspace` directory is shared between the container and the host.

EMUX (ARMX) In The Public

Presentation at Countermeasure 2019 on 7 November 2019. 👇

INSIDE EMUX - Countermeasure 2019 from **Saumil Shah**

Release presentation at HITB+Cyberweek on 16 October 2019. 👇

Introducing EMUX from **Saumil Shah**

Announcing EMUX Docker on 15 June 2021. 👇

Announcing EMUX Docker - DC11332 from **Saumil Shah**

The ARM IoT Firmware Laboratory - NEW TRAINING

> An all new class where the ARM IoT EXPLOIT LABORATORY leaves off. The ARM IoT Firmware Laboratory dives into analysis, extraction and emulation of IoT device firmware, using a variety of techniques. Students shall be given ample hands on practice in emulating a variety of IoT devices. Lab exercises feature firmware extraction directly from the hardware, building a custom kernel and buildroot environment, extracting contents of nvram and emulating the device under EMUX. The class also goes on to fuzzing and exploit development exercises for the emulated devices.

Upcoming classes:

Ringzer0 #VirtualVegas August 2021, Online Remote Training: (4 day class)
https://ringzer0.training/arm-iot-exploitlab.html

Downloads

> The pre-built EMUX PREVIEW VM is now discontinued. You are encouraged to use **EMUX on Docker**
>
> **EMUX Code**
>
> Github: https://github.com/therealsaumil/emux/

EMUX Documentation

- Tutorial: Debugging With EMUX (Github Doc)
- Case Study: Emulating the Tenda AC15 Router (Github Doc)
- Case Study: Extracting the Tenda AC15 Firmware (Github Doc)
- **NEW!** Tutorial: Emulating the D-Link DCS-935L WiFi Camera - MIPS (Github Doc)
- **NEW!** Install guide: Installing EMUX on Kali (Github Doc)

END

- v0.9 22-October-2019, Preview Release
- v1.0 19-November-2019
- v1.1 12-March-2020
- v1.2 05-May-2020
- v1.2 20-May-2020 (minor update)
- v1.3 02-June-2020
- v1.4 11-September-2020
- v2.0 17-June-2021
- v2.1 21-October-2021 Welcome, MIPS! ARMX -> EMUX
- v2.2 29-April-2022