piotr Documentation

Release 1.0.0

Damien Cauquil

Jul 12, 2021

CONTENTS:

CHAPTER

ONE

WHAT IS PIOTR ?

Piotr is a framework designed to create, run, instrument and share virtual devices. It is designed for trainers and security researchers in order to provide an easy way to virtualize an existing device and instrument it with Piotr's API to automate an analysis or to automatically exploit a vulnerability.

Piotr uses Qemu as its emulation core, and especially Qemu's full system emulation for the ARM architecture. It is the only supported architecture so far, but others may be supported in the future depending on Qemu capabilities and evolution.

Piotr is quite similar to Saumil Shah's ARM-X emulation environment, but differs in many ways:

- its architecture is simpler than ARM-X, with no network connection required
- it is really easy to install (even for trainees)
- it provides a convenient way to define virtual devices
- it provides specific tools and a Python API to interact with running virtual devices

1.1 Emulation approach

Piotr follows the same model *ARM-X* previously introduced, relying on a Linux host system that will be used to bootstrap the target environment. The target system runs in a chroot-ed environment inside the host system, thus allowing to debug its processes, access its filesystem without any restriction, etc.

The target device filesystem is mounted over 9P which is a file sharing protocol that does not require any network connection and that is handled by the Linux kernel. No need to host a *samba* server, it works out of the box on any Linux computer.

Virtual devices are defined by a set of files, including a YAML configuration file that tells Piotr how to emulate this device and many more options that may be used to define its behavior.

1.2 Virtual device components

A virtual device is defined by the following components:

- A linux kernel compatible with the original device system
- An optional DTB (*Device-tree block*) file that specifies the internal components and how they are interconnected
- Two filesystems: one for the host and another one for the target device
- A set of scripts that will be used by Piotr to launch the target inside the emulated host system

Piotr manages separately the following components:

- virtual devices definitions (including configuration file, root filesystem and more)
- Host linux kernels that are used by the emulated host system and provides all the required tools to analyse the target system
- Host root filesystems

A stock host Linux kernel and host root filesystem is included in Piotr and automatically installed. These linux kernel and root filesystem provides multiple tools and are designed to automate as much tasks as possible.

Anyway, you may design your own root filesystems or kernels with your own tools and configuration, and install them with Piotr. By doing so, multiple virtual devices may rely on them avoiding redundancy. These kernels and filesystems will be automatically added to exported virtual devices, and installed during importation.

1.3 Piotr for training

As a trainer, I often needed a way to share a virtual device with my trainees. Installing Qemu, configuring it and running a virtual device on a Linux system is far from straightforward, and many trainees had a hard time launching a single virtual device.

Piotr provides a convenient way to export and import virtual devices that will make your life easier. Just make trainees install piotr on their systems, share the virtual device packaged file with them and let them import and run it. That's no more difficult than that, and it saves time.

1.4 Piotr API

Piotr, as a Python-based framework, provides a Python module to interact with a running virtual device and automate various tasks: create and enumerate processes, access its filesystem, or attach a gdbserver to a specific PID. This could be interesting if you want to automate some specific tasks, instruments a virtual device or even automate the exploitation of a vulnerability.

CHAPTER

TWO

SETUP INSTRUCTIONS

2.1 Requirements

Piotr requires *qemu-system-arm* (Full ARM system emulation) in order to work correctly, therefore you must install it before using *Piotr*.

2.1.1 Ubuntu/Debian

```
$ apt install qemu-system-arm
```

2.1.2 Fedora

```
$ dnf install qemu-system-arm
```

2.1.3 ArchLinux

```
$ pacman -S qemu-arch-extra
```

2.2 Install Piotr with pip

You can use *pip* to install Piotr, as shown below:

```
$ pip install piotr
```

2.3 Install from Github

If you want to install the latest version of Piotr from the Github repository, run the following commands:

```
$ git clone https://github.com/virtualabs/piotr.git
$ cd piotr
$ python setup.py install
```

2.4 Additional tools and packages

Avatar2 and *gdb-multiarch* are required if you want to debug a process inside a virtual device from Python. Note that if *Avatar2* is not installed, there is no need to install *gdb-multiarch*.

CHAPTER

THREE

QUICKSTART

This section introduces *Piotr* and demonstrates its basic usages.

3.1 Import an example virtual device

Piotr allows you to import (and export) any virtual device, you can use this feature to install on your system *Saumil Shah*'s Damn Vulnerable ARM Router as shown below:

```
$ wget https://github.com/virtualabs/piotr/blob/main/examples/dvar.piotr?raw=true -O␣
↪dvar.piotr
$ sudo piotr device add dvar.piotr
$ rm dvar.piotr
```

The Damn Vulnerable Arm Router is now available in your virtual devices, as shown with the *device list* command:

```
$ piotr device list

Installed devices:

  > dvar:           Damn Vulnerable ARM Router by Saumil Shah (platform: virt, cpu: -)

 1 device(s) available
```

3.2 Start an instance of Damn Vulnerable ARM Router

Once the Damn Vulnerable ARM Router installed, you can directly launch it with the following command:

```
$ sudo piotr device start dvar
```

sudo is required as Qemu needs some administrative rights to access the network and forward packets from the localhost to the emulated device. You will be prompted with a Piotr banner and will have a shell on the host system. This is the host system, not the emulated device's. You then need to start the services belonging to the emulated device by calling the *target-start* command in the host shell, as shown below:

```
[Host]# target-start
random: fast init done
System ready
Control Server started on port 8080
```

(continues on next page)

(continued from previous page)

```
BusyBox v1.24.2 () built-in shell (ash)

[Guest]#
```

This device listens on port *8081* and *8080*, and you can check it is working by browsing the URL *http://localhost:8080* that will show you something like this:

TODO: add screenshot

In another terminal, you may check that an instance is actually running with the following command:

```
$ sudo piotr device running
sudo piotr device running
Running instances:

  Instance name                          Device
> kind_hofstadter                        Damn Vulnerable ARM Router by Saumil Shah

1 running instance(s)
```

Each running instance is given a random name, unless you specify it when starting a device. The one created here is *kind_hofstadter*, and it identifies this instance. This may be useful if you have multiple running instances.

3.3 Listing instance active processes

Piotr provides also some specific command-line tools that may be helpful. *piotr-ps* allows you to list all the active processes for a given instance:

```
$ sudo piotr-ps
PID   USER     COMMAND
    1 root     init
    2 root     [kthreadd]
    3 root     [rcu_gp]
    4 root     [rcu_par_gp]
    5 root     [kworker/0:0-eve]
    6 root     [kworker/0:0H-kb]
    7 root     [kworker/u2:0-ev]
    8 root     [mm_percpu_wq]
    9 root     [ksoftirqd/0]
   10 root     [rcu_sched]
   11 root     [rcu_bh]
   12 root     [migration/0]
   13 root     [cpuhp/0]
   14 root     [kdevtmpfs]
   15 root     [kworker/u2:1-fl]
   79 root     [kworker/0:1-eve]
  111 root     [khungtaskd]
  257 root     [oom_reaper]
  258 root     [writeback]
  260 root     [kcompactd0]
```

(continues on next page)

(continued from previous page)

```
  261 root     [crypto]
  263 root     [kblockd]
  264 root     [ata_sff]
  381 root     [rpciod]
  382 root     [kworker/u3:0]
  383 root     [xprtiod]
  398 root     [kswapd0]
  469 root     [nfsiod]
  548 root     [kworker/0:1H-kb]
  647 root     [ext4-rsv-conver]
  664 root     /sbin/syslogd -n
  668 root     /sbin/klogd -n
  705 root     /sbin/dhcpcd -f /etc/dhcpcd.conf
  711 root     /usr/bin/qemu-ga -p /dev/vport0p1
  712 root     -sh
  731 root     {target-start} /bin/sh /usr/sbin/target-start
  735 root     {init.sh} /bin/sh /piotr/init.sh
  744 root     /usr/bin/miniweb
  745 root     /usr/bin/lightsrv
  746 root     sh
  767 root     /bin/ps aux
```

3.4 Accessing a pseudo-shell on the emulated device

Piotr provides the *piotr-shell* utility that behaves almost like a *normal* shell except you cannot change directory (a limitation of the current implementation):

```
$ sudo piotr -g -i
>> PIOTR v1.0
>>
>> This is an interactive pseudo-shell with limitations:
>>   - all commands are executed from the root directory
>>   - stderr is not captured and won't be displayed
>>   - no commands history
>>   - no real-time standard output, commands are executed
>>     and results shown once done

[Guest]#
```

3.5 Debugging a remote process with gdb-multiarch

Piotr provides the *piotr-debug* utility that basically runs a *gdbserver* inside the host system and attach it to a given PID:

```
$ sudo piotr-debug 725
Starting gdbserver on the target instance (kind_hofstadter)
Gdbserver is now running on instance with PID 839
```

Once *gdbserver* attached to the target process, you may use *gdb-multiarch* to connect to it and remotely debug the target process.

CHAPTER

FOUR

REFERENCE MANUAL

4.1 Introduction

4.1.1 Why another emulation framework for IoT training and research ?

There are some emulation frameworks available on the Internet that targets devices that embed a Linux operating system (or similar):

- Firmadyne, an automated firmware emulation framework
- ARM-X, a training-oriented emulation framework developed by Saumil Shah

Firmadyne is interesting because it tries to automate everything, but it is also a huge limitation. We did not want an automated system, we were looking for an efficient emulation framework able to run custom-made virtual devices that don't use the same configuration as the original ones.

ARM-X is an emulation designed by Saumil Shah, mostly using shell scripts to provide an emulation environment based on a host system that embeds a target system. The target system runs in its own environment and is therefore easy to analyze. However, the setup is quite complex and adding a new device is somehow challenging for non-experienced users. Moreover, adding or removing virtual devices is not straightforward, and this could be an issue for trainees.

Well, for all these reasons it seemed obvious we needed another Qemu-based instrumentation framework.

4.1.2 Interesting features

Piotr has been designed to be used for IoT security trainings and security research, with the following features in mind:

- full-system qemu-based emulation environment
- easy setup, can be installed with pip
- import/export of virtual devices
- virtual device instances use temporary filesystems to avoid "bricking"
- emphasize on reusability (kernels, filesystems, etc.)
- network should be optional (no samba server, etc…)

4.1.3 Approach

The approach brought by *ARM-X* seems to be efficient and provide the best solution to emulate an embedded device and allow debugging and more at the same time. Piotr definitely relies on an architecture inspired from *ARM-X*.

4.1.4 Supported architectures and platforms

Since Piotr uses *Qemu*, it has the same limitations. Full-system emulation is available in *Qemu* for various architectures, but ARM seems to be the more mature architecture that provides a huge set of different platforms (vexpress, versatilepb, virt, etc.).

Qemu developers recommend to use the *virt* platform as it is the most flexible platform (while others have hardware limitations such as low RAM and limited buses).

At the moment, the only supported architecture is ARM for the *virt* platform.

Note: *Piotr* is designed to run on a Linux system, and has been only tested on this system so far.

4.2 Piotr main concepts

Piotr manages three different types of components:

- host filesystems
- host Linux kernels
- virtual embedded devices

This section describes these components and how they are managed by *Piotr*.

4.2.1 Kernels and host filesystems

Piotr emulated devices rely on an emulated host system to run, and this host system uses a Linux kernel and an associated filesystem. We can design our own different kernels and host filesystems, and add them to Piotr.

More than one virtual device may use the same host kernel and host filesystem, since they are shared amongst virtual devices.

Host filesystem

Piotr host filesystem contains all the required system files required to boot the host system, and some extra tools that are required for analysis:

- *qemu-agent* is provided in Piotr's stock host filesystem and is required by many tools to interact with running instances
- *gdbserver* is provided in Piotr's stock host filesystem to allow remote debugging

Some custom configuration files are also used to display the Piotr banner and allow root to login without password, or even set the prompt according to the running system (host ou emulated target, also called guest).

Filesystems must be ext2 raw images, named as follows: *<platform>-<version string>[-[optional tags]].ext2*. At the moment, only the *virt* platform is supported.

We can use *piotr* to list the registered filesystems:

```
$ piotr fs list
Installed host filesystems:

> virt.cortex-a7.little-5.10.7.ext2   (version 5.10.7, platform: virt, cpu: cortex-a7␣
↪(little-endian), type: ext2)

1 filesystem(s) available
```

Or register a new one:

```
$ piotr fs add virt.cortex-a15.little-5.10.7.ext2
```

Filesystems are usually tied to specific kernel versions as they contain kernel modules that are loaded at boot time.

Our host filesystems are stored in our *Piotr* local folder (*$HOME/.piotr/fs/*), as plain files.

Note: If you don't know what you are doing, just stick with the provided stock filesystem.

Kernel

In the same manner, Piotr manages a list of Linux kernels that would be used to boot the host system but also to run the emulated device. It could be interesting to compile a custom kernel if some features are missing in the stock kernel, or if the emulated device is intended to be run on a specific version.

Kernels are managed the same way the host filesystems are, using *piotr*. We can list the existing kernels by issuing the following command:

```
$ piotr kernel list
Installed kernels:

> virt.cortex-a7.little-5.10.7
Linux version 5.10.7, platform: virt, cpu: cortex-a7 (little-endian)

1 kernel(s) available
```

kernels are named exactly the same way host filesystems are: *<platform>.<cpu>.<endianness>-<version string>*.

We can add or remove kernel with *piotr*, as shown below:

```
$ piotr kernel add virt.cortex-a7.little-5.10.7
$ piotr kernel remove virt.cortex-a7.little-5.10.7
```

Our host kernels are stored in our *Piotr* local folder (*$HOME/.piotr/kernels/*), as plain files.

4.2.2 Virtual embedded device

A virtual embedded device, as Piotr understands it, is a combination of the following:

- a Linux kernel
- a root filesystem
- an (optional) DTB file
- additional files,tools and scripts that are required by Piotr to start the emulated environment

A virtual device is a template that would be used by Piotr to create virtualized environments that mimick a real device behavior.

Here is an example of a device directory:

```
dvar/
  - /config.yaml
  - /rootfs/
```

So, what is a virtual device made of ?

Virtual devices are stored in our *Piotr* local folder (*$HOME/.piotr/devices/*), and each subfolder defines a virtual device.

A virtual device subfolder contains a *config.yaml* file that describes the environment in which the virtual device must run. This configuration file tells *Piotr* how it should configure *Qemu* to correctly emulate the device, by specifying one or more human-readable options.

A root filesystem is also provided (in a specific *rootfs* folder), containing the device root filesystem with the exact permissions and owners. That explains why *Piotr* needs administrative rights to boot a virtual device, as it must access this filesystem and manipulate it. That also means we may need root privileges to browse the content of this root filesystem.

Extra folders may contain a custom linux kernel or a specific DTB file, depending on the device specifications. These files must be referenced in the *config.yaml* file located at the root of the device folder.

Running a virtual device

Piotr does not directly run a device, as it could cause some issues if an unexpected error or mistake is made while it is running. In order to keep the device safe, *Piotr* creates a copy of the host filesystem and the device root filesystem as well, and then runs the device with these copies, avoiding any permanent damage to the original filesystems.

Piotr performs the following task in order to create an instance of a virtual device:

1. it parses the device's *config.yaml* file
2. it checks if a compatible host filesystem and kernel are available
3. it then creates a copy of the host filesystem
4. it launches *qemu-system-arm* with a options that are generated from the configuration
5. when the device boots, it starts all the required services thanks to the host filesystem boot scripts

Any modification brought to the host filesystem during the use of a virtual device won't cause any change. However, any modification brought to the device filesystem will be persistent, except if a specific mode is used to mount this filesystem.

Once a virtual device is running, *Piotr* refers to it as an *instance*. Instances of virtual devices can then be managed the same way as other *Piotr* components do, through *Piotr* command-line utility *piotr*.

To create an instance of a virtual device, use the following command:

```
$ sudo piotr device start dvar
```

It will launch a virtual device from its template, and pick a random instance name. Running instances can be enumerated as follows:

```
$ sudo piotr device running
Running instances:

  Instance name                          Device
> kind_hofstadter                        Damn Vulnerable ARM Router by Saumil Shah

1 running instance(s)
```

We can launch a virtual device with a specific instance name with the following command:

```
$ sudo piotr device start dvar my-dvar-instance
```

And of course, we can stop a running instance with the following command:

```
$ sudo piotr device stop my-dvar-instance
```

When a running instance is stopped, the duplicated host filesystem is removed once the virtual device shut off.

Exporting a virtual device

Piotr provides a way to export a specific virtual device, by packaging all the required dependencies into a single archive file in a way it can be shared and imported.

The packaging process takes the following data and insert them into the archive:

- the device's root filesystem (located in the *rootfs* directory of the device folder, under *$HOME/.piotr/devices/*)
- the device configuration file (*config.yaml*)
- the device kernel (from registered kernels or custom kernel, depending on the device configuration)
- the device host filesystem (from registered host filesystem or custom host file system if defined in the device configuration)

Administrative rights are required in order to export a virtual device.

To export a device, use the *device export* command, as shown below:

```
$ sudo piotr device export davr davr.piotr
```

This command exports a device named *davr* into the *davr.piotr* archive file.

Importing a virtual device

Importing a device basically takes an archive file created by the export feature, and installs everything at the right place:

- the device folder is created in the user Piotr's local folder
- the root filesystem is extracted and stored in plain
- the device kernel is installed and registered if it is not one dedicated to this device
- the host filesystem is installed and registered if it is not one dedicated to this device

The device is then ready to use, with all of its dependencies automatically installed. Kernel files and host filesystems installed and registered during import may be used to create new devices as well.

To import a device, use the *device add* command as shown below:

```
$ sudo piotr device add davr.piotr
```

Virtualizing an existing embedded device

If we plan to virtualize an embedded device, there are a few steps to follow. Each of these steps can fail for one reason or another, so there is no certainty that we would be able to virtualize a specific device:

- we must determine the version of its Linux kernel and the specific drivers it uses
- we must have a copy of the root filesystem of the device we want to emulate
- we must also determine how the system accesses (read/write) its non-volatile parameters

Extracting the root filesystem is not straightforward, and in most cases it is split among multiple partitions that we would have to assemble to recover the actual root filesystem. Doing so would also mean modifying some configuration files or bootup scripts that are used to mount everything at the right place.

Identifying the version of the kernel used by the device, as well as the custom drivers that should be loaded in order for the system to boot correctly may be challenging, depending on the system. Again, we would have to find some tricks to avoid using these drivers, when sometimes we would end up coding some fake drivers to make the system believes everything is normal while it is obviously not the case. It is sometimes better to stick to the expected linux kernel version, even if it causes some issues to the emulated host.

4.3 Creating a virtual device

Creating a virtual device from an existing real device requires to:

- extract or rebuild its root filesystem
- identify the underlying hardware (CPU, memory, etc.)

4.3.1 Create a device template

Piotr provides a command to create a default device template:

```
$ sudo piotr device create my-device
```

This will create a folder named *my-device* in your *piotr* device directory (i.e. *~/.piotr/devices/*), and populate it with a default configuration file and an empty *rootfs* directory.

The device directory should look like this:

```
device/
    config.yaml
    rootfs/
```

4.3.2 Rebuild the device's root filesystem

The main idea is to rebuild the device's root filesystem including its mounted partitions. For instance, if your device mounts */dev/mtdblock0* to */usr*, we have to manually copy the files present in the partition filesystem into */usr*.

Thus, we end up with a link-free filesystem similar to the one used by the device when it runs. This filesystem must be copied in the *rootfs* directory, in the corresponding device directory. We must perform this step as root, as we need to keep the correct permissions, user and group IDs in this filesystem.

4.3.3 Create the device configuration file

Once the root filesystem ready, we need to fill the *config.yaml* file present in the device directory.

First, we set the target architecture (based on what was observed on the real hardware), as shown below:

```
version: "1.0"
device:
    name: My IoT device
    machine:
        platform: virt
        memory: 1024M
        cpu: cortex-a7
```

The configuration above declares a device called "My IoT device" that will run on Qemu's virt platform (the only currently supported by *piotr*), with 1024M of RAM.

We then tells *piotr* which kernel to use and how to load the device root filesystem:

```
version: "1.0"
device:
    name: My IoT device
    machine:
        platform: virt
        memory: 1024M
        cpu: cortex-a7

    kernel: 4.19.196
    bootargs: "root=/dev/vda rw console=ttyAMA,115200"
    guestfs: virtfs
```

We tell *piotr* to use a generic Linux kernel 4.14.131 (that ships with the latest version of *piotr*), we also provide some boot arguments (*bootargs*) which are pretty standard for Qemu's virt platform, and asks for our device root filesystem to be loaded through Plan9 Resource Sharing protocol (9P2000). This last option can be omitted as it is the default behavior.

However, you may want *piotr* not to use this sharing mechanism and therefore use *embed* instead of *virtfs*. In this case, *piotr* will use a copy of the device root filesystem and embed it into the host filesystem before running it.

We have specified so far the machine architecture, hardware platform and the kernel to use (with its boot arguments). We may want to ask *piotr* to forward a TCP port to access our device SSH service for instance, through the following configuration:

```
version: "1.0"
device:
    name: My IoT device
    machine:
        platform: virt
        memory: 1024M
        cpu: cortex-a7

    kernel: 4.19.196
    bootargs: "root=/dev/vda rw console=ttyAMA,115200"
    guestfs: virtfs

    network:
        nic0: user

    redirect:
        nic0:
            ssh: tcp,2222,22
```

4.3.4 Creating a bootup script

When our emulated host will start our device in a *chrooted* environment, it will execute a specific script to start the device's services. This script will act as an init script, without all the mountings and device specific tasks that will not work as expected, since it is absolutely not the real hardware.

This script must be located in a *piotr* folder in the device root filesystem, and called *init.sh*. Below an example of such a script:

```
#!/bin/sh

# Emulate sdcard (required if you are using the sdcard option in config.yaml)
mount -t ext2 /dev/vdb /mnt/sdcard

# Add devpts support (mandatory)
mount devpts /dev/pts -t devpts

# Start prerun program
# (required to avoid errors due to emulation)
/mnt/mtd/prerun

# Set the guest shell prompt
export PS1='[Guest]# '
```

(continues on next page)

(continued from previous page)

```
# Start a shell
sh
```

4.3.5 Booting your device

When we are done with the root filesystem, device configuration file and init script, we can give our emulated IoT device a try. We use *piotr* to start the device and the emulated host system boots up:

```
$ sudo piotr device start ipcam
Booting Linux on physical CPU 0x0
[...]
Serial: AMBA PL011 UART driver
9000000.pl011: ttyAMA0 at MMIO 0x9000000 (irq = 54, base_baud = 0) is a PL011 rev1
console [ttyAMA0] enabled
SCSI subsystem initialized
[...]
NET: Registered protocol family 17
9pnet: Installing 9P2000 support
Registering SWP/SWPB emulation handler
rtc-pl031 9010000.pl031: setting system clock to 2021-06-30 08:58:11 UTC (1625043491)
ALSA device list:
No soundcards found.
EXT4-fs (vda): mounted filesystem without journal. Opts: (null)
VFS: Mounted root (ext4 filesystem) on device 254:0.
devtmpfs: mounted
Freeing unused kernel memory: 1024K
Run /sbin/init as init process
EXT4-fs (vda): re-mounted. Opts: (null)
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Saving random seed: random: dd: uninitialized urandom read (512 bytes read)
OK
Starting network: OK
Starting dhcpcd...
no interfaces have a carrier
forked to background, child pid 713
Starting ser2net: no configuration file
Starting qemu agent...
random: dhcpcd: uninitialized urandom read (120 bytes read)









    -----< version 1.0.0 >-----
```

(continues on next page)

(continued from previous page)

```
[Host]#

We then start the guest (our embedded device):
```

```
[Host]# target-start
random: fast init done
Can't open /dev/akgpio
: No such file or directory
GPIO dev not init!!!
=== Start no-auth telnetd server ===
open /dev/akpcm_cdev0 failed: No such file or directory.
=== play type : 0 ===
GPIO dev not init!!!
Can't open /dev/akgpio
: No such file or directory
otg_hs: version magic '3.4.35 mod_unload ARMv5 ' should be '4.19.91 SMP mod_unload ARMv7␣
↪p2v8 '
insmod: can't insert '/mvs/modules/otg-hs.ko': invalid module format
[Guest]#
```

A single device cannot be found (*/dev/akgpio*) and some drivers could not be loaded due to a wrong kernel version used to start the target system, but it boots up and runs all the network services we want to test.

We may also compile a Linux kernel for the exact same architecture and create a compatible host filesystem. As one can see, Linux version 3.4.35 is required here.

However, emulating real hardware such as GPIOs or even a CCCD sensor will be very difficult and this demonstrates the limits of emulation.

4.4 Using a custom kernel and host root filesystem

As shown above, some devices may require dedicated configurations that do not fit the standard use. In this case, it is recommended to create a kernel specifically for a device, along with a compatible host filesystem.

4.4.1 Prerequisites

We need a framework to build a kernel and a root filesystem: *buildroot*. *Buildroot* provides a very convenient way to compile kernels and create a root filesystem.

It is usually available in the main Linux distributions application repositories, or can be downloaded from its website (https://buildroot.org).

4.4.2 Building a kernel for Piotr

If we need a specific kernel version for a virtual device, we will build one that fits our needs with buildroot. This section is not intended to be a complete guide for *buildroot*, but will cover the specifics required to compile a kernel compatible with Piotr.

In order to use all the required features, *buildroot*'s toolchain must support WCHAR and C++.

Kernel configuration

First, you must configure *buildroot* to build a compatible kernel and filesystem for a Qemu ARM architecture compatible with Qemu's *virt* platform:

```
$ make qemu_arm_versatile_defconfig
```

For recent Linux kernel versions, Plan 9 resource sharing support (9P2000) must be enabled. In *buildroot*, the kernel configuration is done through a text-based interface:

```
$ make linux-menuconfig
```

First, enable *Plan 9 Resource Sharing Support* in *Networking support*. Then in *Filesystems* > *Network File Systems*, make sure *Plan 9 Resource Sharing Support (9P2000)* is enabled. 9P POSIX ACLs or security labels may be enabled, but are not mandatory.

By default, *buildroot* enables all the required options for Qemu, and it would do the job for recent versions of Linux kernel. If you plan to use older versions of Linux kernel, it may be challenging to get buildroot to compile it as it may require older versions of gcc that may be incompatible. Moreover, please consider using the *embed* option in your YAML device configuration file for option *device.guestfs* rather than *virtfs*.

Compilation

Once your kernel configured, run the following command to compile it:

```
$ make linux
```

Buildroot will compile the selected kernel version and will produce a *zImage* file in the *output/images/* folder. Rename this file as follows:

```
$ mv output/images/zImage /tmp/virt.cortex-a7.little-5.10.7
```

The expected pattern is *platform.cpu.endianness-x.y.z*, you must comply with it in order to be able to register/install this kernel into Piotr's kernels.

Installation

Use *piotr* to install your kernel. It will be copied into Piotr's kernels folder and automatically available.

```
$ sudo piotr kernel add /tmp/virt.cortex-a7.little-5.10.7
```

4.4.3 Building a root filesystem

Using *buildroot*, it is possible to create a root filesystem that provides everything required to host our target device filesystem.

Mandatory tools required by Piotr

Buildroot allows the following applications to be built and installed in the target root filesystem, under the *Target packages* submenu when configuring *buildroot*:

```
$ make menuconfig
```

- *gdb* and *gdbserver* (requires a toolchain that supports c++, wchar_t, threads and thread debugging)
- Qemu guest agent (*qemu-ga*) provided by the "Qemu tools" package

These are mandatory, but we may also install for convenience:

- nano as a text editor
- filesystems utilities (squashfs, e2fsprogs, etc.)
- python3

Note: Use the same toolchain as you would do for the corresponding kernel, in order to build executable files that will run under the target architecture !

Creating the root filesystem

Once *buildroot* configured, just use *make* to build the filesystem:

```
$ make
```

The generated filesystem is available in *output/images/rootfs.ext2* and is ready to be modified, because we need to add Piotr's host filesystem files.

We mount this filesystem on a mountpoint, and then add the required files:

```
# mkdir /tmp/fs
# mount -t ext2 ./output/images/rootfs.ext2 /tmp/fs
# cp -rf <piotr dir>/hostfs-template/* /tmp/fs/
# umount /tmp/fs
```

Eventually, we rename this root filesystem following the expected pattern:

```
# mv ./output/images/rootfs.ext2 /tmp/virt.cortex-a7.little-1.0.0.ext2
```

And we add it to our stock host filesystem using *piotr*:

```
$ sudo piotr add /tmp/virt.cortex-a7.little-1.0.0.ext2
```

And this host filesystem is then installed and available.

4.5 API

Since *Piotr* is a python-framework, it exposes an API that can be used to automate tasks such as:

- starting and stopping an instance of a virtual device
- executing commands on the emulated host system or the target that runs in it
- enumerating processes on the emulated device
- debugging remote processes

4.5.1 Importing Piotr API

Piotr API is imported in Python with the following code:

```
from piotr.api import *
```

4.5.2 Creating and accessing a virtual device

Starting a virtual device

To create and start an instance of a virtual device, we must first get a *Device* object corresponding to the device we want to instanciate:

```
device = Device('dvar')
```

Then, we can create an instance by calling *run()* as shown below:

```
instance = device.run(alias='my-instance', background=True)
```

The call to *run()* returns an *Instance* object that represents the running virtual device.

Retrieving a running instance

Piotr allows us to enumerate the running instances, by using *Piotr* and its *instances()* method:

```
for instance in Piotr.instances():
    print(instance)
```

A specific running instance can be retrieved with its instance name, using *instance()*:

```
instance = Piotr.instance('my-instance')
```

4.5.3 Managing processes

Creating a process

We can create a process that will run inside the host system by calling *exec_host()*, as shown below:

```
# Launch /usr/bin/example on host system and in background
pid = instance.exec_host('/usr/bin/example', wait=False)
```

We may also want to start a process in the context of the target system, by using *exec_target()*:

```
# Launch /usr/bin/example on target system, and in background
pid = instance.exec_target('/usr/bin/example', wait=False)
```

Enumerating processes

It is then possible to enumerate the running processes on this instance:

```
for process in instance.ps():
    print('PID:%d - %s' % (process.pid, process.path))
```

Finding a process PID

If we want to find the PID of an executable based on its path, use *pid()* method with the search executable path:

```
pid = instance.pid('/usr/bin/example')
```

Since target and host processes are available from the host system, we do not have to specify on which system the process we are looking for is ran.

Terminating a process

To terminate a process, just call the *kill()* method as shown below:

```
pid = instance.pid('/usr/bin/example')
if pid is not None and pid>0:
    instance.kill(pid)
```

4.5.4 Remote debugging a process

It is also possible to attach a *gdbserver* to a process running in the host or target system, and then to interact with this server. First, we need to debug a specific running process:

```
target_pid = instance.pid('/bin/my-target-program')
if target_pid > 0:
    dbg = instance.debug(target_pid)
```

The *debug()* method starts a *gdbserver* instance, attach it to the target process, and returns a `Debugger` object. This object drives a *gdb* debugger and allows to:

- access the remote process registers

- access the remote process memory
- set and remove breakpoints
- run, single step and stop execution

Note: This debugger capability requires avatar2 to be installed on our machine, as it uses a component provided by this Python package. This package is not installed by default, but is mandatory for this feature.

Controlling the execution

Once our debugger attached, the process is stopped. We can set a breakpoint at a specific address:

```
# Set breakpoint at address 0x11e8
dbg.set_breakpoint(0x11e8)

# Continue execution
dbg.cont()

# Wait for breakpoint to be reached
dbg.wait()
```

Accessing and modifying registers

Registers can be read with the `read_register()` method, and written with the `write_register()` method:

```
# Show PC
print('PC: 0x%08x' % dbg.read_register('pc'))

# Modify PC
dbg.write_register('pc', 0x11f4)

# Continue execution
dbg.cont()
```

4.5.5 Stopping a running instance

Just call the *stop()* method to stop a running instance:

```
instance.stop()
```

CHAPTER

FIVE

API REFERENCE

Piotr API

This module allows to interact with/instrument Piotr virtual devices in a pythonic way. It exposes a main class, *Piotr*, that is able to interact with running instances and to automate things.

class piotr.api.**Device**(*deviceName*)
API Device object.

get_sysroot()
Return this device root fs path

run(*alias=None*, *background=False*)
Run emulated device.

exception piotr.api.**DeviceNotFound**

class piotr.api.**Instance**(*guest*, *device*)
This class represents a Piotr running device instance and allows to interact with it, execute commands into the emulated host and target (if Qemu agent is supported by the device), enumerate and manipulate process.

Parameters

- **guest** (*piotr.qemu.QemuPlatform*) – Qemu guest
- **device** (*str*) – Name of parent device, as referenced in Piotr devices list

debug(*pid*, *ip='0.0.0.0'*, *port=4444*, *gdb_executable='gdb-multiarch'*)
Runs a gdb server and attaches it to a process.

exec_host(*command*, *wait=True*)
Execute a command in host.

exec_target(*command*, *wait=True*)
Execute a command in guest.

Parameters

- **command** (*str*) – command to execute
- **return_output** (*bool*) – wait for the process to end and return output

Returns command output or executable PID

get_sysroot()
Return this instance root path

kill(*pid*, *sig=9*)
Kill a processus.

pid(*process_name*)
Find process ID from process name :param process_name: process name :type process_name: str :return: PID of the process, None on error

ps()
Run *ps* on this target and return a list of processes.

Returns list of processes

stop()
Stop instance.

target_start()
Run target in host.

class piotr.api.**Piotr**
Piotr main API.

static devices()
Enumerate registered devices.

static instance(*inst_name*)
Find an existing instance by name

static instances()
Enumerate running instances.

class piotr.api.**Process**(*pid=- 1*, *user=None*, *path=None*)
This class holds information about a process on the emulated system.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX