# Linux debugging, profiling, tracing and performance analysis training

## Practical Labs

bootlin
https://bootlin.com

June 30, 2025

## About this document

Updates to this document can be found on https://bootlin.com/doc/training/debugging.

This document was generated from LaTeX sources found on https://github.com/bootlin/training-materials.

More details about our training sessions can be found on https://bootlin.com/training.

## Copying this document

# Training setup

*Download files and directories used in practical labs*

## Install lab data

For the different labs in this course, your instructor has prepared a set of data (kernel images, kernel configurations, root filesystems and more). Download and extract its tarball from a terminal:

```
$ cd
$ wget https://bootlin.com/doc/training/debugging/debugging-labs.tar.xz
$ tar xvf debugging-labs.tar.xz
```

Lab data are now available in an `debugging-labs` directory in your home directory. This directory contains directories and files used in the various practical labs. It will also be used as working space, in particular to keep generated files separate when needed.

## Update your distribution

To avoid any issue installing packages during the practical labs, you should apply the latest updates to the packages in your distro:

```
$ sudo apt update
$ sudo apt dist-upgrade
```

You are now ready to start the real practical labs!

## Install extra packages

Feel free to install other packages you may need for your development environment. In particular, we recommend to install your favorite text editor and configure it to your taste. The favorite text editors of embedded Linux developers are of course *Vim* and *Emacs*, but there are also plenty of other possibilities, such as Visual Studio Code[1], *GEdit*, *Qt Creator*, *CodeBlocks*, *Geany*, etc.

It is worth mentioning that by default, Ubuntu comes with a very limited version of the `vi` editor. So if you would like to use `vi`, we recommend to use the more featureful version by installing the `vim` package.

## More guidelines

Can be useful throughout any of the labs

- Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.
- Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.
- Never stay stuck with a strange problem more than 5 minutes. Show your problem to your colleagues or to the instructor.
- You should only use the `root` user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.

[1] This tool from Microsoft is Open Source! To try it on Ubuntu: `sudo snap install code --classic`

 

- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command to give the new files back to your regular user.
  Example: `$ sudo chown -R myuser.myuser linux/`

 

# Preparing the system

*Objectives:*

- *Prepare the STM32MP157D board*

## Install needed packages

You need some development packages before being able to build the target firmware:

```
$ sudo apt install build-essential git
```

## Building the image

We created a special image for the training. This image will contain everything we need (tools, configured kernel, etc). This image will be built with buildroot which allows to build a complete image for embedded systems.

```
$ cd /home/$USER/debugging-labs/
$ git clone https://github.com/bootlin/buildroot
$ cd buildroot
$ git checkout debugging-training/2025.02
$ make stm32mp157a_dk1_debugging_defconfig
$ make
```

This will take a few minutes. At the end of the build, the `output/images/` directory will contain the images that can be used on the board. During this course, we will use a kernel located on a sdcard and a rootfs via NFS. This will let us transfer data freely from and to the target board.

The rootfs should be extracted at `/home/$USER/debugging-labs/nfsroot` using this command:

```
$ tar xvf output/images/rootfs.tar -C /home/$USER/debugging-labs/nfsroot
```

We will also export the `CROSS_COMPILE` variable to set the toolchain as our cross compiling toolchain:

```
$ export CROSS_COMPILE=/home/$USER/debugging-labs/buildroot/output/host/bin/arm-linux-
```

This export needs to be either done in each shell in which `CROSS_COMPILE` is going to be used or added to your shell configuration (`.bashrc` for instance)

## Flashing the sdcard

In order to get a working board, you will need to flash a sdcard with the `output/images/sdcard.img` file. Plug your sdcard on your computer and check on which `/dev/sdX` it has been mounted (you can use the `dmesg` command to check that). For instance, if the sdcard has been mounted on `/dev/sde`, use the following command:

```
$ sudo dd if=output/images/sdcard.img of=/dev/sde
$ sync
```

NOTE: Double-check that you are targeting the correct device before executing the dd command!

Once flashed, insert the sdcard into the STM32MP157D board.

 

## Prepare the STM32MP1 Discovery Kit 1

The STM32MP1 Discovery Kit 1 is powered via a USB-C cable, which you need to connect to the `CN6` (also labeled `PWR_IN`) connector.

In addition, to access the debug serial console, you need to use a micro-USB cable connected to the `CN11` (also labeled `ST-LINK`) connector.

Once your micro-USB cable is connected, a `/dev/ttyACM0` device will apear on your PC. You can see this device appear by looking at the output of `dmesg` on your workstation.

To communicate with the board through the serial port, install a serial communication program, such as `picocom`:

```
sudo apt install picocom
```

If you run `ls -l /dev/ttyACM0`, you can also see that only `root` and users belonging to the `dialout` group have read and write access to this file. Therefore, you need to add your user to the `dialout` group:

```
sudo adduser $USER dialout
```

**Important**: for the group change to be effective, in Ubuntu 18.04, you have to *completely reboot* the system [2]. A workaround is to run `newgrp dialout`, but it is not global. You have to run it in each terminal.

Now, you can run `picocom -b 115200 /dev/ttyACM0`, to start serial communication on `/dev/ttyACM0`, with a baudrate of `115200`. If you wish to exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

There should be nothing on the serial line so far, as the board is not powered up yet.

## Set up the Ethernet communication on the workstation

With a network cable, connect the Ethernet port of your board to the one of your computer. If your computer already has a wired connection to the network, your instructor will provide you with a USB Ethernet adapter. A new network interface should appear on your Linux system.

Find the name of this interface by typing:

```
$ ip a
```

The network interface name is likely to be `enxxx`[3]. If you have a pluggable Ethernet device, it's easy to identify as it's the one that shows up after pluging in the device.

Then, instead of configuring the host IP address from NetworkManager's graphical interface, let's do it through its command line interface, which is so much easier to use:

```
$ nmcli con add type ethernet ifname en<xxx> ip4 192.168.0.1/24
```

## Setting up the NFS server

Install the NFS server by installing the `nfs-kernel-server` package:

```
$ sudo apt install nfs-kernel-server
```

Once installed, edit the `/etc/exports` file as `root` to add the following lines, assuming that the IP address of your board will be `192.168.0.100`:

```
/home/<user>/debugging-labs/nfsroot 192.168.0.100(rw,no_root_squash,no_subtree_check)
```

[2] As explained on https://askubuntu.com/questions/1045993/after-adding-a-group-logoutlogin-is-not-enough-in-18-04/.

[3] Following the *Predictable Network Interface Names* convention: https://www.freedesktop.org/wiki/Software/systemd/PredictableNetworkInterfaceNames/

 

Of course, replace `<user>` by your actual user name.

Make sure that the path and the options are on the same line. Also make sure that there is no space between the IP address and the NFS options, otherwise default options will be used for this IP address, causing your root filesystem to be read-only.

Then, restart the NFS server:

```
$ sudo exportfs -r
```

If there is any error message, this usually means that there was a syntax error in the `/etc/exports` file. Don't proceed until these errors disappear.

## U-Boot setup

In order to use a rootfs on NFS, we will use an external rootfs. This can be specified by passing bootargs to the kernel. To do so, we are going to set the `bootargs` U-Boot variable and save the environment. To be able to edit U-Boot variables, we need to interrupt its standard boot sequence: with the serial console opened, maintain Enter key while pressing once the reset button, and wait for U-Boot prompt to appear. Then, enter the following commands:

```
STM32MP1> env set bootargs root=/dev/nfs ip=192.168.0.100:::::eth0
  nfsroot=192.168.0.1:/home/<user>/debugging-labs/nfsroot/,nfsvers=3,tcp rw
STM32MP1> saveenv
```

NOTE: Be sure to replace the `<user>` string with the correct user.

## Login

You can now run the system by pressing the RESET button on the board.

You can login on the serial console or via SSH (`ssh root@192.168.0.100`). The username is `root` and the password is `root`.

 

# System status

*Objectives:*

- *Observe running processes using* ps *and* top*.*
- *Check memory allocation and mapping with* procfs *and* pmap*.*
- *Monitor other resources usage using* iostat*,* vmstat *and* netstat*.*

## Observe system status

In order to examine the platform, you can either execute commands through the `picocom` terminal or open a SSH connection.

Now that the board is up and running, let's try to understand what is running on this system. The provided image includes numerous tools to analyze the system. Try to answer the following questions using the commands that were presented during the course:

1. How many CPU does this processor have?
2. What are the memory maps used by the `dropbear` process?
3. How much PSS memory is used by `dropbear`?
4. What is the amount of memory available for applications on the system?
5. How much unused memory is left on our system?
6. Is there swapped memory?
7. Is there an application using too much CPU?
8. How much time is spent by CPU0 in system mode (kernel)?
9. Is there some IOs ongoing with storage devices?
10. How many Mbytes/s are transferred from/to the MMC card?
11. What is the process generating transfers to the MMC?
12. Which processor receives most of the interrupts?
13. How many interrupts were received from the MMC controller?

Once found, you can remove the files `/etc/init.d/S25stress-ng` and `/etc/init.d/S26mmc-reader` and reboot to have a cleaner system.

 

# Solving an application crash

*Objectives:*

- *Analysis of compiled C code with compiler-explorer to understand optimizations.*
- *Managing* gdb *from the command line.*
- *Debugging a crashed application using a coredump with* gdb.
- *Using* gdb *Python scripting capabilities.*
- *(Bonus) configuring an IDE for debugging*

## Compiler explorer

Go to https://godbolt.org/ and paste the content of `/home/$USER/debugging-labs/nfsroot/root/compiler_explorer/swap_bytes.c`. Select the correct compiler for armv7 and observe the generated assembly. Try to modify the compiler options to optimize the generation (-O3). Observe the result.

## Using GDB

Take our `linked_list.c` program. It uses the `<sys/queue.h>` header which provides multiple linked-list implementations. This program creates and fill a linked list with the names read from a file. Compile it from your development host using the following command:

```
$ cd /home/$USER/debugging-labs/nfsroot/root/gdb/
$ make
```

By default, it will look for a `word_list` file located in the current directory. This program, when run on the target, should display the list of words that were read from the file.

```
$ ./linked_list
```

From what you can see, it actually crashes! So we will use GDB to debug that program. We will do that remotely since our target does not embed a full gdb, only a gdbserver, a lightweight gdb server that allows connecting with a remote full feature GDB. Start our program on the target using gdbserver in multi mode:

```
$ gdbserver --multi :2000 ./linked_list
```

On the host side install gdb-multiarch if not already done and attach to this process using gdb-multiarch:

```
$ sudo apt install gdb-multiarch
$ gdb-multiarch ./linked_list
(gdb) target extended-remote 192.168.0.100:2000
(gdb) set sysroot /home/<user>/debugging-labs/buildroot/output/staging/
```

Then continue the execution and try to find the error using GDB. There are multiple ways to debug such program. Here are a few hints to help you find the bug root cause:

 

- Instead of waiting for the segfault to happen, you can use breakpoints to stop at a specific place BEFORE the segfault happens
- The bug is pretty reproductible here, so do not hesitate to restart the program multiple times during your analysis to properly understand the path leading to the bug, and to run it step by step
- If your breakpoint is in a loop, it may be tedious to type "continue" multiple times to reach to execution site you are interested in: remember that you can use conditional breakpoints and watchpoints to make a breakpoint hit on a specific loop occurrence!
- It is necessary to understand the data structures your code is using to understand why you eventually get a segfault: take some time to get familiar with the `SLIST_HEAD` and `SLIST_ENTRY` macros, and to print them during debugging to see how data is organized.
  - You can also refer to the `macro` command documentation to learn how to interpret macros in gdb during the program execution
- You mave even use and call existing code during your debug session: you can for example try to print the list by executing `display_linked_list()` with the `p` gdb command!

*TIP: you can execute command automatically at GDB startup by putting them in a* `~/.gdbinit` *file. For instance, history can be enabled with* `set history save on` *and pretty printing of structure with* `set print pretty on`*.*

*TIP: GDB features a TUI which can be spawned using Ctrl + X + A. You can switch from the command line to the TUI view using Ctrl X + O.*

*NOTE: you can exit gdbserver from the connected gdb process using the* `monitor exit` *command.*

## Using a coredump with GDB

Sometimes, the problems only arise in production and you can only gather data once the application crashed. This is also something that can be used if the crash is not reproducible but crashes only once in a while. If so, we can use the kernel coredump support to generate a core dump of the faulty application and do a post-mortem analysis.

First of all, we need to enable kernel coredumping support of programs. On the target, run:

```
$ ulimit -c unlimited
```

Then, run the program normally:

```
$ ./linked_list
Segmentation fault (core dumped)
```

When crashing, a `core` file will be generated. On your desktop computer, fix the permissions using:

```
$ sudo chown $USER:$USER core
```

Then, open this file using `gdb-multiarch`:

```
$ gdb-multiarch ./linked_list ./core
```

You can then inspect the program state (memory, registers, etc) at the time it crashed. While less dynamic, it allows to pinpoint the place that triggered the crash. Here are a few hints to help you manipulate the coredump:

- Since the coredump is only a memory snapshot (the program is not running anymore), you can not run any process-controlling command in gdb (`run`, `continue`, `step`...)

 

- However, you can print any variable which is in scope. You can even define your own commands to ease printing our custom list:
  - Use the `define` command to start defining a command, use `end` to mark its end
  - In between, knowing how the data is organized, you can print some data fields with `p`
  - You can use gdb custom variables to navigate into the list: refresh your custom variable to make it point to the next element of the list each time you call your custom command

## GDB Python support

When developing and debugging applications, sometimes we often uses the same set of commands over and over under GDB. Rather than doing so, we can create python scripts that are integrated with GDB.

In order to display our program list from GDB, we provide a python GDB script named `linked_list.py` that displays this list. You will need to fill two parts of this script to display a complete list correctly. This python script takes the list head name and the next field name as parameters.

The part to be filled in are the pretty printer struct formatting and the iteration on the list. We would like to display each `struct name` as `index: name`. In order to access a struct field in gdb python, you can use `self.val['field_name']`.

Once done, you can use the following commands in your development host gdb client session to test your script:

```
(gdb) source linked_list.py
(gdb) printslist name_list next
```

## (Bonus) Configuring an IDE for debugging

While using bare gdb commands in console is enough to debug your application issues, you may want to use some frontend to do so, for example by using GDB features directly in your IDE. Multiple frontends are able to interact with GDB thanks to its MI interface For this example, we will configure Visual Studio Code. Start by installing the IDE.

The firmware used for this training labs is in fact already running a gdbserver right at boot, listening on port 9876: thanks to the `--multi` option, this server can be configured by the client to debug any target of interest.

We must then provide instructions to VSCode about how to start a debug session. You may have noticed that the labs data also provide a `.vscode` directory along the program sources. If you open vscode in the sources directory, it will recognize this `.vscode` directory and load corresponding configuration

The `.vscode` directory contains a minimalistic configuration to allow remote debugging, thanks to two files:

- `launch.json` contains a debug configuration which indicates what is the program to debug, how to connect to gdbserver and to configure it, where to find the missing symbols (our staging directory), etc.
- `tasks.json` contains a default build task: we have defined in our debug configuration that we want to run a build before each debug session (so we are sure that the program under debug is in sync with our sources), so we have to define this build task. Note that since we are using a NFS root, this build task also takes in charge the ”deploy” step!

You can then open VSCode in the sources directory:

```
cd ~/debugging-labs/root/gdb
code .
```

Before being able to debug your program onto the target, you will need to install the official C/C++ extension (you can do it directly from the ”Extensions” tab in the left column).

You can now start the debugging session (hit the ”Run and Debug” tab on the left column, then ”Debug linked list” in the list on top left, or directly hit ”F5” to start the debug session). VSCode will start our cross-gdb and connect it to the remote gdbserver, and you will be able to do everything we have done in the console so far:

- Add breakpoints by clicking on the line number you want to break in
- Set conditions on breakpoint by issuing right-click on the breakpoint (red circle on the line number), ”Edit breakpoint”
- In the debugging tab (left column), you can
  - Check variables which are in scope
  - take a look at your call stack
  - add expression to watch for specific values
- Control the program execution with the ”Continue”, ”Step Over”, ”Step Into”, ”Step Out”, ”Restart”, ”Stop” buttons (in the upper right section)
- Use directly bare gdb commands in the debugging console: open the ”Debug console” tab on the lower part of the screen, then type any command learnt so far, but remember to prefix it with `-exec`

You can also refer to VSCode debug configuration documentation to learn more about available configuration options

 

# Application tracing

*Objectives:*

- *Analyze dynamic library calls from an application using* ltrace.
- *Using* strace *to analyze program syscalls.*

## ltrace

On your computer, go into the `ltrace` lab folder:

```
$ cd /home/$USER/debugging-labs/nfsroot/root/ltrace/
$ make
```

Next, run the `authent` application on the target.

```
$ cd /root/ltrace
$ export LD_LIBRARY_PATH=$PWD
$ ./authent
Error: failed to authenticate the user !
```

Note: Since our application uses a local dynamic shared library which is not in the default paths expected by ld (see `man 8 ld.so`), we need to provide that path using `LD_LIBRARY_PATH`.

As you can see, it seems our application is failing to correctly authenticate the system. Using *ltrace*, trace the application on the target in order to understand what is going on.

```
$ ltrace ./authent
```

From that trace, try to find which function fails.

In order to overload this check, we can use a `LD_PRELOAD` a library. We'll override the `al_authent_user()` based on the `authent_library.h` definitions. Create a file `overload.c` which override the `al_authent_user()`, prints the user, password and returns 0. Compile it on your development host using the following command line:

```
$ ${CROSS_COMPILE}gcc -fPIC -shared overload.c -o overload.so
```

Finally, run your application and preload the new library using the following command on the target:

```
$ LD_PRELOAD=./overload.so ./authent
```

## strace

`strace` is useful to debug an application when you don't have the source. For that example, use the `strace_me` binary that is present in on the target in `/root/strace` and run it with strace:

```
$ cd /root/strace
$ strace ./strace_me
```

Based on the output and running strace with other options, try to answer the following questions:

 

- What are the files that are opened by this binary?
- How many time is `read()` called?
- Which `openat` system calls failed?
- How many system calls are issued by the program?

 

# Debugging memory issues

*Objectives:*

- *Memory leak and misbehavior detection with* valgrind *and* vgdb.

## valgrind & vgdb

On your development host, go into the `valgrind folder` and compile `valgrind.c` with debugging information using:

```
$ cd /home/$USER/debugging-labs/nfsroot/root/valgrind
$ make
```

Then run it on the target. Do you notice any problem? Does it run correctly? Even though there is no segfault, an application might have some memory leaks or even out-of-bounds accesses, uninitialized memory, etc.

Now, run the command again with valgrind using the following command:

```
$ valgrind --leak-check=full ./faulty_mem_app
```

You'll see various errors found by valgrind

- Invalid memory write
- Uninitialized memory
- Memory leaks

In order to pinpoint exactly each error and be able to debug with gdb, vgdb can be used. We will do that remotely on the host using gdb-multiarch. First, we need to run valgrind with vgdb enabled on the target:

```
$ cd /root/valgrind
$ valgrind --vgdb=yes --vgdb-error=0 --leak-check=full ./faulty_mem_app
```

Then, in order to do remote debugging, we also need to run vgdb in listen mode. Start another terminal in SSH on the target and run the following command:

```
$ vgdb --port=1234
```

On the computer side, start gdb-multiarch and give it the `faulty_mem_app` binary which will allow to detect the architecture and read symbols:

```
$ gdb-multiarch ./faulty_mem_app
```

Finally, we'll need to connect to vgdb using the following gdb command:

```
(gdb) target remote 192.168.0.100:1234
```

You will then be able to debug each error using gdb and valgrind will interrupt the program each time it detects an error. Try to solve all the problems that were encountered by valgrind.

NOTE: The backtrace for leaks is not shown on the target because all libraries are stripped and thus do not have any debugging symbols anymore. This leads to the impossibility to use the dwarf information for

 

backtracing.

 

# Application profiling

*Objectives:*

- *Visualizing application heap using* Massif.
- *Profiling an application with* Cachegrind, Callgrind *and* KCachegrind.
- *Analyzing application performance with* perf.

## Massif

Massif is really helpful to understand what is going on with the memory allocation side of an application. Compile the `heap_profile` example that we did provide using the following command on your development host

```
$ cd /home/$USER/debugging-labs/nfsroot/root/heap_profile
$ make
```

Once compiled, run it on the target under massif using the following command:

```
$ cd /root/heap_profile
$ valgrind --tool=massif --time-unit=B ./heap_profile
```

NOTE: we use `--time-unit=B` to set the X axis to be based on the allocated size.

Once done, a `massif.out.<pid>` file will be created. This file can be displayed with `ms_print`. Based on the result, can you answer the following questions:

- What is the peak allocation size of this program?
- How much memory was allocated during the program lifetime?
- Do we have memory leaks at the end of execution?

You can also visualize the data collected by Massif with a graphical interface thanks to `massif-visualizer`. To do so, execute the following commands on your development host:

```
$ cd /home/$USER/debugging-labs/nfsroot/root/heap_profile
$ sudo apt install massif-visualizer
$ massif-visualizer massif.out.*
```

Note: `heaptrack` is not available on buildroot but is available on debian. You can try the same experience using heaptrack on your computer and visualizing the results with `heaptrack_gui`.

## Cachegrind & Callgrind

Cachegrind and Callgrind allow to profile a userspace application by simulating the processor that will run it. In order to analyze our application and understand where time is spent, we are going to profile it with both tools.

Let's start by profiling the application using the `cachegrind` tool. Our program takes two file names as parameters: an input PNG image and an output one. We provided a sample image in `tux_small.png` which can be used as an input file. First let's compile it using the following commands on our development host:

 

```
$ cd /home/$USER/debugging-labs/nfsroot/root/app_profiling
$ make
```

We are going to profile cache usage using Cachegrind with the following command on the target:

```
$ valgrind --tool=cachegrind --cache-sim=yes ./png_convert tux_small.png out.png
```

The execution will take some time and a `cachegrind.out.<pid>` will be generated. Once finished, on the host, fix the permissions on the `cachegrind.out.*` file to be able to open it with `Kcachegrind`:

```
$ sudo chown $USER:$USER cachegrind.out.*
```

Analyze the results with `Kcachegrind` in order to understand the function that generates most of the D cache miss time.

Based on that result, modify the program to be more cache efficient. Run again the cachegrind analysis to check that the modifications were actually effective.

We also profile the execution time using callgrind by running valgrind again on the target but with a different tool:

```
$ valgrind --tool=callgrind ./png_convert tux_small.png out.png
```

Again, on the host platform, fix the permissions of the file using:

```
$ sudo chown $USER:$USER callgrind.out.*
```

Again, analyze the results using `Kcachegrind`. This time, the view is different and allows to display all the call graphs.

Looking at the results, it seems like our conversion function is actually taking a negligible time. However, valgrind simulate the program with an ”ideal” cache. In real life, the processor is often used by other applications and the kernel also takes some time to execute which leads to cache disturbance.

## Perf

In order to have a better view of the performance of our program in a real system, we will use `perf`. In order to gather performance counter from the hardware, we will run our program using `perf stat`. We would like to observe the number of L1 data cache store misses. In order to select the correct event, use `perf list` on the target to find it amongst the cache events:

```
$ perf list cache
```

Once found, execute the program on the target using `perf stat` and specify that event using `-e`:

```
$ perf stat -e L1-dcache-store-misses ./png_convert tux.png out.png
```

Revert the modifications that we did to invert the program loops and again, measure the amount of misses.

After that, we will record our program execution using the `perf record` command to obtain a callgrind like result.

```
$ perf record ./png_convert tux_small.png out.png
```

Once recorded, a `perf.data` file will be generated. This file will contain the traces that have been recorded. These traces can be analyzed using `perf report` on the development host:

```
$ sudo chown $USER:$USER perf.data
$ perf report \
```

 

```
        --symfs=/home/$USER/debugging-labs/buildroot/output/staging/ \
        -k /home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/vmlinux
```

You will quickly notice that the output is not the same as valgrind because it displays the time spent per function (excluding function calls inside them). This allows to find which function takes most of the execution time. In order to compare this output to the valgrind one, we can run perf and also record the callgraph using the `--call-graph` option.

```
$ perf record --call-graph dwarf ./png_convert tux_small.png out.png
```

We specify that we want to record the call graph using the DWARF information that are contained in ELF file (compiled with `-g`). Once recorded, on the desktop platform, display the results with `perf report` and compare them with callgrind ones on your development platform.

*NOTE: the vmlinux file used for the* `-k` *option must match the kernel build id that is running on the board or perf will not use it.*

*NOTE: in order to annotate the disassembled code and display the time spent for each instruction,* `CROSS_COMPILE` *must be set.*

 

# System wide profiling and tracing

*Objectives:*

- *System profiling with* trace-cmd *and* KernelShark.
- *(Bonus) System profiling with* perf *and FlameGraphs.*

## ftrace & uprobes

First of all, we will start a small program on the target using the following command:

```
$ mystery_program 1500 400 2 &
```

In order to trace a full system, we can use ftrace. However, if we want to trace the userspace, we'll need to add new tracepoints using uprobes. This can be done manually with the uprobe sysfs interface or using `perf probe`.

Before starting to profile, we will compile our program to be instrumented. On your development host, run:

```
$ cd /home/$USER/debugging-labs/nfsroot/root/system_profiling
$ make
```

On the target, we will create a uprobe in the main function of the `crc_random` program each time a crc is computed. First, let's list the line numbers that are recognized by perf to add a uprobe:

```
$ cd /root/system_profiling
$ perf probe --source=. -x ./crc_random -L main
```

*Notes:*

- *In order to be able to add such userspace probe, perf needs to access symbols and source file*
- *If perf output is filled with misinterpreted ANSI escape sequences, you can append* `| cat` *to the command*

Then, we can create a uprobe and capture the crc value using:

```
$ perf probe --source=. -x ./crc_random main:35 crc
```

We can finally trace the application using `trace-cmd` with this event. We will use the remote tracing capabilities of `trace-cmd` to do that. First, we will start the `trace-cmd` server on the desktop platform:

```
trace-cmd listen -p 4567
```

Then, on the target, run `trace-cmd` with the `-N` parameter to specify the remote trace-cmd server. Interrupt it after about 50 CRC computations in order to have a meaningful trace.

```
$ trace-cmd record -N 192.168.0.1:4567 -e probe_crc_random:main_L35 ./crc_random
^C
```

*Note: if during the recording you encounter an* `Unsupported file version 7` *error, it likely means that the trace-cmd tool on your workstation is older than the one on the target, and does not understand the default trace format used by the target. You can force trace-cmd on the target to use an older version of the trace data format by also providing* `--file-version=6` *on the recording command*

 

Then, using KernelShark on the host, analyze the trace:

```
$ sudo apt install kernelshark
$ kernelshark
```

We can see that something is wrong, our process does not seem to compute crc at a fixed rate. Let's trace the `sched_switch` events to see what is happening:

```
$ trace-cmd record -N 192.168.0.1:4567 -e sched_switch -e probe_crc_random:main_L35 ./crc_\
    random
^C
```

Reanalyze the traces with KernelShark and try to understand what is going on.

## (Bonus) System profiling with *perf* and FlameGraphs

In order to profile the whole system, we are going to use perf and try to find the function that takes most of the time executing.

First of all, we will run a global recording of functions and their backtrace on (all CPUs) during 10 seconds using the following command on the target:

```
$ perf record -F 99 -g -- sleep 10
```

This command will generate a `perf.data` file that can be used on a remote computer for further analysis. Copy that file and fix the permissions using `chown`:

```
$ cd /home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/
$ sudo cp /home/$USER/debugging-labs/nfsroot/root/system_profiling/perf.data .
$ sudo chown $USER:$USER perf.data
```

We will then use perf report to visualize the aquired data:

```
$ perf report --symfs=/home/$USER/debugging-labs/buildroot/output/staging/
       -k /home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/vmlinux
```

Another useful tool for performance analysis is flamegraphs. Latest perf version includes a builtin support for flamegraphs but the template is not available on debian so we will use another support provided by Brendan Gregg scripts.

```
$ git clone https://github.com/brendangregg/FlameGraph.git
$ perf script | ./FlameGraph/stackcollapse-perf.pl | ./FlameGraph/flamegraph.pl > flame.html
```

Using this flamegraph, analyze the system load.

*TIP: if the generated graph is not relevant/missing symbols, you may try to to all symbols translation in the target before generating the graph on host, i.e use perf script on the target, and bring back the result on host and provide it to Flamegraph scripts*

You can also generate a Flamegraph on previous labs: for example you can re-do a recording on `png_convert` from the application profiling lab, generate the corresponding Flamegraph and confirm your observations about most time-consuming functions.

 

# eBPF tooling with BCC

*Objectives:*

- *Use BCC to easily create a custom tracing tool*

Let's assume we have doubts about what is running on our system at any point of time, and that we do not have the appropriate tools to perform the corresponding analysis. We will then develop our own tooling with eBPF to trace any new program executed on the system.

We will first build a quick prototype with BCC and run it directly onto **our development machine** (ie not the target). Install the `bcc` package onto your host, and create a `trace_programs.py` script. Write the necessary code to trace any new program executed on the whole system. You have to take care of the following points:

- we want a small program whose sole purpose is to print the following message in the ftrace buffer each time a new program is executed: `New process <PID> running program <COMM>`.
- To manage to capture the relevant data, we want to attach our program on the entrypoint of the `execve` syscall. One way to do so is to create a kprobe on the corresponding syscall handler in the kernel, and to attach our eBPF program onto it. Check the BCC reference guide to learn how to hook BCC programs to kprobes (there are at least two different ways).
- You will need to get the exact prototype of the `execve` entrypoint function you want to hook to get its arguments. Also, depending on which architecture we are running, the name of the function to target may not be the same. There are multiple ways to find the exact function prototype:
  - you can check `man 2 execve` to get `execve` arguments, but you will still lack the exact entrypoint name
  - you can search for any function related to `execve` in `/sys/kernel/tracing/available_filter_functions` to find the entrypoint name.
- To build our log line, we need to capture both the process ID and the executable name in our eBPF program:
  - To fetch the PID of the calling process you can use kernel-provided bpf helpers (see `man 7 bpf-helpers`)
  - Executable filename is one of the `sys_execve` function arguments, so your eBPF program should receive it as an argument as well, since it is a kprobe-type program
- Also make sure that your program does not finish after loading and attaching your program, or it will be released immediately: you can for example write a busy loop calling the BCC method `trace_print` to directly print ftrace buffer from your script

Once done, start you script, open a new console and run a few commands, you should be able to trace all those commands with your script

*Notes:*

- *Since eBPF subsystem needs root priviledges to be manipulated, you need sudo to run the script*
- *If you have made some mistakes in your eBPF program, the verifier will refuse to load it, or worse, it will not even build. BCC makes sure to print the relevant logs to ease debugging, so make sure to read and understand those.*

 

# eBPF tooling with libbpf

*Objectives:*

- *Port our BCC tool onto the target thanks to libbpf*
- *Implement new features in our program*

Unfortunately, embedding BCC scripts onto our target is not very convenient: we need to bring python, llvm, clang... So it may be more relevant to switch our tool to libbpf. Before starting converting our tool, make sure that the following packages are installed on your development system:

- `clang` to be able to build bpf programs
- `linux-tools-common` to get `bpftool` (needed to generate skeletons)
- `libbpf-dev` to get access to `libbpf` APIs in our eBPF program
- `pahole` to allow BTF header (vmlinux.h) generation with bpftool

The first step is to prepare our bpf program:

- Go to the labs directory, in `ebpf/libbpf` directory. In there, you will find `trace_programs.bpf.c`. It is the exact same eBPF program as the one used in the BCC script, but any BCC-specific API or macro has been replaced with libbpf functions or macros. Take some time to spot and understand the differences with the previous version:
  - This program may access some kernel structures at some point, so it has been prepared to benefit from CO-RE (to remain compatible between different kernel versions), that's why it depends on a `vmlinux.h` header that we will have to generate.
  - In order to manipulate kprobes, the program needs some libbpf header, and because the data manipulated by kprobes changes with the platform (it directly uses registers), we need to define the target architecture with `__TARGET_ARCH_arm`
  - The code uses the `SEC` macro to place the eBPF program in a specific section: libbpf will use this section to learn about the program type and attach point
  - It also uses the `BPF_KPROBE` macro to allow to get access to already-interpreted arguments from the kprobes: without this macro, we would have to identify the relevant registers to parse the targeted function arguments
  - Be careful that the `bpf_trace_printk` is not the same helper as the one used with BCC, and so the way to call it is slightly different
- You will first need to generate the vmlinux header used in the eBPF program. You can use bpftool to do so:

```
$ bpftool btf dump file /home/${USER}/debugging-labs/buildroot/output/build/linux-\
    6.6.21/vmlinux format c > vmlinux.h
```

- Next you need to build your eBPF program into a loadable object:

```
$ clang -target bpf -g -O2 -c trace_programs.bpf.c -o trace_programs.bpf.o
```

*Note: since our program deals with pt_regs, it is not portable between architectures, that's why we have to provide the target architecture with* `__TARGET_ARCH_arm`

Your eBPF program does not really need a userspace program to run, and the data it emits can be read from the trace buffer. So to use it in a minimal way you can use `bpftool` without writing a userspace tool.

However you'll also need to use the bpf filesystem to keep a reference to the loaded program, or it will be unloaded as soon as bpftool exits:

```
mount -t bpf none /sys/fs/bpf
mkdir /sys/fs/bpf/myprog
bpftool prog loadall trace_programs.bpf.o /sys/fs/bpf/myprog autoattach
```

Display the ftrace buffer content:

```
bpftool prog tracelog
```

Open another console onto the target (through SSH) and execute some programs: they should appear in the tracing buffer.

Wait for at least a minute. Did your tracer allow you to spot anything suspect?

Feel free to experiment using bpftool to better understand how your eBPF program is managed by the kernel, for example using `bpftool prog`:

```
bpftool prog
bpftool prog dump xlated id <id>
```

Before continuing, ensure you remove the eBPF program from the kernel:

```
rm -fr /sys/fs/bpf/myprog/
```

Managing the eBPF program with bpftool is a simple way to experiment with it. However a user-friendly tool should not require all the manual steps with bpftool and the bpf filesystem. By building on top of the work done so far, add a userspace program to automate loading the eBPF program with a single command:

- Generate a C skeleton header from this object with bpftool and libbpf

  ```
  $ bpftool gen skeleton trace_programs.bpf.o name trace_programs > trace_programs.skel.h
  ```

  Check the generated header: you will see that the raw bpf program has been embedded in the header, but also that you have a small set of APIs available to easily design your tracing tool.

You now have to write the userspace part in charge of managing your eBPF program:

- Create a `trace_programs.c` file. In there, include your freshly created skeleton header, create a main function, and use the available APIs to open, load and attach your program. You can refer to the kernel documentation to learn how to use those skeleton APIs: `bpf/libbpf/libbpf_overview`
- Once again, remember to make sure that your userspace program does not end after attaching your eBPF program, otherwise it will be detached and unloaded immediately. You can add a busy loop in your code to prevent it.
- libbpf expects you to "destroy" ebpf objects when you are done using it, check your skeleton file to find the relevant API.

Finally, build your program:

```
$ ${CROSS_COMPILE}gcc trace_programs.c -lbpf -o trace_programs
```

Run your tracing tool on the target. Now you don't need to use bpftool and the bpf filesystem anymore. Your users will be glad!

## Improving our program

Now that we have a working base for our custom tracing tool, we will improve it to make it more useful.

 

In the labs directory, go to `ebpf/libbpf_advanced`. Copy the `trace_programs.c` and `trace_programs.bpf.c` from the previous part in this directory, as you will iterate on it. The directory provides a makefile which automates all build steps performed manually earlier. To use this makefile, make sure to have your `CROSS_COMPILE` variable properly set, as well as a `KDIR` variable pointing to your kernerl directory:

```
export KDIR=/home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21
```

Having to open ftrace to display the logs is cumbersome, we would like to get the trace directly in the console in which we have started the tool. We will use the opportunity to switch our program output from a log line in ftrace to events pushed in a perf ring buffer. A perf ring buffer is a kind of map which can be used in eBPF programs to stream data to userspace in a very efficient way. To use a perf ring buffer, perform the following steps:

- Edit your eBPF program to push data into a perf ring buffer instead of ftrace:
  - Create a structure type containing the data we will push in the ring buffer. This struct will contain two pieces of information for now: a PID, and a program name. Since you will need to use this structure from both the eBPF program and the userspace program, define it in a shared header.
  - Create the map in your eBPF program file. There are different ways of defining maps in eBPF programs, we will create a BTF-defined map:

```
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 32);
} rb SEC(".maps");
```

  - Edit your program code to push data in the ring buffer each time it is triggered: create an instance of your custom data structure in the BPF program, fill it with the event information and push it into the buffer.
    - You can not use strcpy or memcpy in your program to copy the executable name in your event structure, you have to use the bpf helper `bpf_probe_read_str`.
    - To push the custom data structure into the perf ring buffer, you can use another bpf helper called `bpf_ringbuf_output`.
- Finally, edit your userspace program to retrieve data from the perf ring buffer, thanks to libbpf APIs
  - Now that we have added a map into our program, the skeleton object has a handle to this map in its `maps` field
  - To manipulate the ring buffer in the userspace program, you have access to specific libbpf APIs, especially `ring_buffer__new` to create an instance of the ring buffer, and `ring_buffer__poll` to poll the buffer in your main loop. Unfortunately, the official documentation is quite succinct on those functions, but you can take a look at the `tools/testing/selftests/bpf/` directory in the kernel source tree to learn how to use those (search for `ring_buffer__new()` and `ring_buffer__poll()` usage in Elixir)
  - You may need to convert maps objects into the corresponding file descriptors. libbpf also provide APIs to do so.
  - In the event callback passed to `ring_buffer__new`, retrieve the data from the ring buffer and print it.

Once done, run your updated program onto the target: you should see some traces directly in the console in which you have started the tracing tool.

As a final improvement, we will trace the parent PID as well to know who is starting any program.

- Edit your eBPF program to read the parent PID. This info can be captured by retrieving the current `struct task_struct`, and identifying the relevant fields. Check both Elixir for the layout of `struct task_struct`, and `man 7 bpf-helpers` to learn how to get the current task.
- We are using CO-RE definition for kernel data (through vmlinux.h), so we can not dereference directly a `struct task_struct` in our eBPF program, we must use helpers to retrieve struct fields. You can check this blog post from Andrii Nakryiko to learn about such helpers: you will need to use either `bpf_core_read` function or even the `BPF_CORE_READ` macro, both availables from the `bpf/bpf_core_read.h` header from libbpf. Also, you will need to check `struct task_struct` to know what field to extract to get the parent PID.
- Update your userspace program to read and print the newly captured value

Once done, run your script again, you can now see the parent process of any new program executed on the target, and so investigate further any suspicious activity on the system!

 

# Kernel debugging: built-in testing

*Objectives:*

- *Debugging locks and sleeps mistakes using* PROVE_LOCKING *and* DEBUG_ATOMIC_SLEEP *options.*
- *Find a module memory leak using* kmemleak.

## Locking and sleeps problems

CONFIG_PROVE_LOCKING and CONFIG_DEBUG_ATOMIC_SLEEP have been enabled in the provided kernel image. First, compile the module on your development host using the following command line:

```
$ cd /home/$USER/debugging-labs/nfsroot/root/locking
$ export CROSS_COMPILE=/home/$USER/debugging-labs/buildroot/output/host/bin/arm-linux-
$ export ARCH=arm
$ export KDIR=/home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/
$ make
```

On the target, load the `locking.ko` module and look at the output in dmesg:

```
# cd /root/locking
# insmod locking_test.ko
# dmesg
```

Once analyzed, unload the module. Try to understand and fix all the problems that have been reported by the `lockdep` system.

## Kmemleak

The provided kernel image contains kmemleak but it is disabled by default to avoid having a large overhead. In order to enable it, reboot the target and enable kmemleak by adding `kmemleak=on` on the command line. Interrupt U-Boot at reboot and modify the `bootargs` variable:

```
STM32MP> env edit bootargs
STM32MP> <existing bootargs> kmemleak=on
STM32MP> boot
```

Then compile the dummy test module on your development host:

```
$ cd /home/$USER/debugging-labs/nfsroot/root/kmemleak
$ export CROSS_COMPILE=/home/$USER/debugging-labs/buildroot/output/host/bin/arm-linux-
$ export ARCH=arm
$ export KDIR=/home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/
$ make
```

On the target, load the `kmemleak_test.ko` and trigger an immediate kmemleak scan using:

```
# cd /root/kmemleak
# insmod kmemleak_test.ko
```

 

```
# rmmod kmemleak_test
# echo scan > /sys/kernel/debug/kmemleak
```

Note that you might need to run the `scan` command several times before it detects a leakage due to memory still containing references to the leaked pointer. Soon after that, the kernel will report that some leaks have been identified. Display them and analyze them using:

```
# cat /sys/kernel/debug/kmemleak
```

You will see that the symbols addresses do not make sense. This is due to the `kptr_restrict` configuration which must be change to allow displaying pointer addresses. To do so, use the following command on the target:

```
# sysctl kernel.kptr_restrict=1
```

You can use `addr2line` to identify the location in source code of the lines that did cause the reports. You may need to substract module loading address: you can guess the address by taking a look at `/proc/modules` while the module is loaded.

You may also notice other memory leaks that are actually some real memory leaks that did exist in the kernel version used for this training !

Once the lab is done, don't forget to remove `kmemleak=on` from your kernel commandline.

 

# Kernel debugging: OOPS analysis and KGDB

*Objectives:*

- *Analyzing an* oops.
- *Debugging with* KGDB.

## OOPS analysis

We noticed that the watchdog command generated a crash on the kernel. In order to reproduce the crash, run the following command on the target:

```
$ watchdog -T 10 -t 5 /dev/watchdog0
```

Immediatly after executing this commands, you'll see that the kernel will report an OOPS!

### Analyzing the crash message

Analyze the crash message carefully. Knowing that on ARM, the `PC` register contains the location of the instruction being executed, find in which function does the crash happen, and what the function call stack is.

Using Elixir (https://elixir.bootlin.com/linux/latest/source) or the kernel source code, have a look at the definition of this function. In most cases, a careful review of the driver source code is enough to understand the issue. But not in that case!

### Locating the exact line where the error happens

Even if you already found out which instruction caused the crash, it's useful to use information in the crash report.

If you look again, the report tells you at what offset in the function this happens. We will disassemble the code for this function to understand exactly where the issue happened.

That is where we need a kernel compiled with `CONFIG_DEBUG_INFO`, which is already enabled in the kernel we compiled in the initial lab. This way, the kernel vmlinux file is compiled with `-g` compiler flag, which adds a lot of debugging information (matching between source code lines and assembly for instance).

Using `addr2line`, find the exact source code line were the crash happened. For that, you can use the following command on your development host:

```
$ addr2line -e /home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/vmlinux \
       -a <crash_address>
```

This can also be done automatically using `decode_stacktrace.sh`. First, copy/paste the OOPS message into the `~/debugging-labs/oops.txt` file. Then, using the script provided by the kernel, execute the following command:

```
$ cd /home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/
$ export ARCH=arm
$ export CROSS_COMPILE=/home/$USER/debugging-labs/buildroot/output/host/bin/arm-linux-
$ ./scripts/decode_stacktrace.sh vmlinux < ~/debugging-labs/oops.txt
```

We can even go a step further and use the cross GDB to open vmlinux and locate the function and corresponding offset in assembly

```
$ ${CROSS_COMPILE}gdb /home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/vmlinux
(gdb) disassemble <function>
```

## KGDB debugging

In order to debug this OOPS, we'll use KGDB which is an in-kernel debugger. The provided image already contains the necessary KGDB support and the watchdog has been disabled to avoid rebooting while debugging. In order to use KGDB and the console simultaneously, compile and run kdmx on your development host:

```
$ cd /home/$USER/debugging-labs
$ git clone https://git.kernel.org/pub/scm/utils/kernel/kgdb/agent-proxy.git
$ cd agent-proxy/kdmx
$ make
$ ./kdmx -n -d -p/dev/ttyACM0 -b115200
serial port: /dev/ttyACM0
Initalizing the serial port to 115200 8n1
/dev/pts/7 is slave pty for terminal emulator
/dev/pts/8 is slave pty for gdb

Use <ctrl>C to terminate program
```

Note: the slave ports number will depend on the run.

**Important: before using `/dev/pts/7` and `/dev/pts/8`, the picocom process that did open `/dev/ttyACM0` must be closed!**

On the target, setup KGDB by setting the console to be used for that purpose in kgdboc module parameters:

```
$ echo ttySTM0 > /sys/module/kgdboc/parameters/kgdboc
```

Once done, trigger the crash by running the watchdog command, the system will automatically wait for a debugger to be attached. Run the cross GDB and attach a gdb process to KGDB with the following command:

```
$ ${CROSS_COMPILE}gdb ${HOME}/debugging-labs/buildroot/output/build/linux-6.6.21/vmlinux
(gdb) target remote /dev/pts/8
```

*TIP: in order to allow auto-loading of python scripts, you can add* `set auto-load safe-path /` *in your .gdbinit file*

First of all, confirm with GDB the information that were previously obtained post-crash. This will allow you to also display variables values. Starting from that point, we will add a breakpoint on the `watchdog_set_drvdata()` function. However, this function is called early in boot so we will need to actually attach with KGDB at boot time. To do so, we'll modify the bootargs to specify that. In U-Boot, add the following arguments to bootargs using `env edit`:

```
STM32MP> env edit bootargs
STM32MP> <existing bootargs> kgdboc=ttySTM0,115200 kgdbwait
STM32MP> boot
```

 

Then the kernel will halt during boot waiting for a GDB process to be attached. Attach gdb client from your development host using the same command that was previously used:

```
$ ${CROSS_COMPILE}gdb /home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/vmlinux
(gdb) target remote /dev/pts/8
```

Note: if you prefer using gdb-multiarch instead of the cross-gdb we have built and do not specify a file to be used, gdb-multiarch won't be able to detect the architecture automatically and the target command will fail. In that case, you can set the architecture using:

```
(gdb) set arch arm
(gdb) set gnutarget elf32-littlearm
```

Before continuing the execution, add a breakpoint on `watchdog_set_drvdata()` using the `break` GDB command and then continue the execution using the `continue` command

```
(gdb) break watchdog_set_drvdata
(gdb) continue
```

Analyze the subsequent calls and find the place where the driver data are clobbered.

TIP: you can fix the problem in "live" by modifying the content of the `wdd->driver_data` variable directly using the following command:

```
(gdb) p/x var=hex_value
```

Use it to set the variable with the previous value that was used before getting clobbered with NULL. Once done, continue the execution and verify that you fixed the problem using the `watchdog` command.

*Note: In theory, we could have added a watchpoint to watch the address that was modified but the ARM32 platform does not provide watchpoint support with KGDB.*

## Debugging a module

KGDB also allows to debug modules and thanks to the GDB python scripts (`lx-symbols`) mainly, it is as easy as debugging kernel core code. In order to test that feature, we are going to compile a test module and break on it. On your development host, build the module:

```
$ cd /home/$USER/debugging-labs/nfsroot/root/kgdb
$ export CROSS_COMPILE=/home/$USER/debugging-labs/buildroot/output/host/bin/arm-linux-
$ export ARCH=arm
$ export KDIR=/home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/
$ make
```

Then on the target, insert the module using insmod:

```
# cd /root/kgdb
# insmod kgdb_test.ko
```

We can now enter KGDB mode and attach the external gdb to it. We will do that using the magic SySrq 'g' key. Before that, ensure the tty device for gdb already set, or set it now:

```
# echo ttySTM0 > /sys/module/kgdboc/parameters/kgdboc
# echo g > /proc/sysrq-trigger
```

The kernel will then enter KGDB mode and will wait for a gdb connection. On the development platform, start it and attach to the target:

```
$ ${CROSS_COMPILE}gdb /home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/vmlinux
(gdb) target remote /dev/pts/8
```

Then you will need to execute the `lx-symbols` command in gdb to reload the symbols from the module. You'll also need to pass a list of path that contains the external modules:

```
(gdb) lx-symbols /home/${USER}/debugging-labs/nfsroot/root/kgdb/
loading vmlinux
scanning for modules in /home/<user>/debugging-labs/nfsroot/root
loading @0xbf000000: /home/<user>/debugging-labs/nfsroot/root/kgdb/kgdb_test.ko
```

*NOTE: If KGDB were already connected and the lx scripts loaded, then* `lx-symbols` *would be run automatically on module loading.*

Finally, add a breakpoint right after the `pr_debug()` call and continue the execution to trigger it:

```
(gdb) break kgdb_test.c:17
(gdb) continue
```

At some point, the breakpoint will be triggered. Try to display the variable `i` to display the current loop value.

Note: Due to a GDB bug, sometimes, gdb will crash when continuing. You can use a temporary breakpoint using the gdb `tbreak` command to workaround this problem.

Bonus: as a side quest you can try to enable the `pr_debug()` call using the dynamic debug feature of the kernel. This can be done using the `/proc/dynamic_debug/control` file.

 

# Kernel debugging: post-mortem analysis with kexec & kdump

*Objectives:*

- *Setting up* Kexec & kdump.
- *Extracting a coredump for a crashed kernel.*

## kdump & kexec

As presented in the course, kdump/kexec allows to boot a new kernel and dump a perfect copy of the crashed kernel (memory, registers, etc) which can be then debugged using gdb or crash.

### Building the dump-capture kernel

We will now build the dump-capture kernel which will be booted on crash using kexec. For that, we will use a simple buildroot image with a builtin initramfs using the following commands on the development host:

```
$ cd /home/$USER/debugging-labs/buildroot
$ make O=build_kexec stm32mp157a_dk1_debugging_kexec_defconfig
$ make O=build_kexec
```

Then, we'll copy the zImage and the device-tree to the nfs /root/kexec directory:

```
$ mkdir /home/$USER/debugging-labs/nfsroot/root/kexec
$ cp build_kexec/images/zImage /home/$USER/debugging-labs/nfsroot/root/kexec
$ cp build_kexec/images/stm32mp157a-dk1.dtb /home/$USER/debugging-labs/nfsroot/root/kexec
```

These files are now ready to be used from the target using kexec.

### Configuring kexec

First of all we need to setup a kexec suitable memory zone for our crash kernel image. This is achieved via the linux command line. Reboot, interrupt U-Boot and add the `crashkernel=60M` parameter. This will tell the kernel to reserve 60M of memory to load a "rescue" kernel that will be booted on panic. We will also add an option which will panic the kernel on oops to allow executing the kexec kernel.

```
STM32MP> env edit bootargs
STM32MP> <existing bootargs> crashkernel=60M oops=panic
STM32MP> boot
```

To load the crash kernel into the previously reserved memory zone, run the following command on the target:

```
# kexec --type zImage -p /root/kexec/zImage --dtb=/root/kexec/stm32mp157a-dk1.dtb
  --command-line="console=ttySTM0,115200n8 maxcpus=1 reset_devices"
```

Once done, you can trigger a crash using the previously mentioned watchdog command:

 

```
$ watchdog -T 10 -t 5 /dev/watchdog0
```

At this moment, the kernel will reboot into a new kernel using the specified kernel after displaying the backtrace and a message:

```
[ 1181.987971] Loading crashdump kernel...
[ 1181.990839] Bye!
```

After reboot, log into the new kernel normally and bring up the eth0 interface: (We will use 192.168.0.101 to avoid cloberring ssh `know_hosts` file for the 192.168.0.100 entry).

```
$ ifconfig eth0 192.168.0.101
```

**Note:** ethernet setup might timeout due to some init issues after kexec boot so this commands needs to be run another time.

```
$ cd /home/$USER/debugging-labs/
$ scp root@192.168.0.101:/proc/vmcore .
```

You can load the `vmcore` file using cross GDB:

```
$ ${CROSS_COMPILE}gdb /home/$USER/debugging-labs/buildroot/output/build/linux-6.6.21/vmlinux \
    vmcore
```

 