Get Started with Pip

September 23, 2021

Contents

1 Setting up your development environment

To get started with Pip, it is required to install the appropriate development environment. This section describes the tools required by Pip as well as the three ways to obtain a functional development environment.

1.1 Required tools

Coq Proof Assistant: Pip's source code and formal proof of its memory isolation properties are written using the Coq proof assistant. In order to compile Coq files and generate the required intermediate files for the kernel to build, you will need the 8.13.1 version of Coq. A proper way to install Coq is via opam.

Doxygen: Pip's documentation is generated through CoqDoc (included with Coq) for the Coq part, and Doxygen for the C part. The documentation is not mandatory to compile Pip, but it is highly required that you compile it and keep it somewhere safe so you always have some reference to read if you need some information about Pip's internals.

GNU C Compiler: GCC is the only C compiler known to compile Pip correctly. CLANG, for example, is not yet supported. To that end, you need a version of GCC capable of producing 32bits ELF binaries.

GNU Debugger: The GNU Debugger allows you to debug a partition while it is executed on the top of Pip. This is very useful during the development process. That's the reason why GDB is not mandatory but highly recommended.

GNU Make: Althought Pip is known to compile on FreeBSD and OSX hosts, these need a GNU software in order to perform the compilation, which is GNU Make 4.3 and above.

GNU GRUB: GNU GRUB is a boot loader which allows to create bootable ISO file. It is not mandatory but required if you want to produce a bootable ISO file of your project.

Netwide Assembler: Pip's assembly sources for the x86 architecture are assembled using the Netwide Assembler (NASM). A known working version is version 2.14, although any version since 2.0 should be working.

OCaml Package Manager: Opam is the package manager for the OCaml programming language, the language in which Coq is implemented. This is the proper way to install and pin the Coq Proof Assistant to a specific version.

TeX Live TeX Live is an open source TeX distribution required to generate the *getting started* of Pip. It not mandatory, but required if you want to generate this document.

QEMU: Although it is not required to build Pip, it is highly recommended to run Pip on emulated hardware rather than physical hardware during development. As such, QEMU is a known, multi-platform emulator, and is fully integrated into Pip's toolchain.

Haskell Stack: Pip uses a home-made extractor to convert Coq code into C code. In order to compile this Extractor, which is written in Haskell, we use the Stack tool to download and install automatically the required GHC and libraries.

1.2 Virtual machine image

Before starting, you need to install a virtualization software such as VirtualBox or VMware. You can follow the procedure on their websites.

Once the installation is completed, you need to download the archived OVA image of the virtual machine and the SHA-256 message digest:

```
# Download the archived OVA image of the virtual machine
$ wget http://pip.univ-lille1.fr/image/vm/pip.tar.gz

# Download the SHA-256 message digest of the archived image
$ wget http://pip.univ-lille1.fr/image/vm/pip.tar.gz.
    sha256sum
```

When the download is complete, you can check the integrity of the downloaded file:

```
$ sha256sum -c pip.tar.gz.sha256sum
```

Now, you have to extract the archived image:

```
$ tar -xvf pip.tar.gz
```

Once the extraction is complete, you have to import the OVA image into the virtualization software, then start the virtual machine.

The login credentials are:

```
Login: pip
Password: pip
```

or

```
Login: root
Password: pip
```

Your development environment is ready.

1.3 Docker image

Before starting, you need to install Docker on your machine. You can follow the procedure on their website. Once the installation is completed, you have to download the archived Docker image and the SHA-256 message digest:

```
# Download the archived Docker image of Pip
$ wget http://pip.univ-lille1.fr/image/docker/pip.tar.gz

# Download the SHA-256 message digest of the archived image
$ wget http://pip.univ-lille1.fr/image/docker/pip.tar.gz.
    sha256sum
```

When the download is complete, you can check the integrity of the downloaded file:

```
$ sha256sum -c pip.tar.gz.sha256sum
```

Now, you need to import the archived image:

```
$ docker load -i pip.tar.gz
```

and check that it is imported:

```
$ docker image ls
```

Once the Docker image imported, you can either run a new container from the image in interactive mode:

```
# Run Pip's image inside of a new container
$ docker run -it --name pip pip bash

# Run a command in the running container
$ whoami

# Exit the shell
$ exit
```

or in detached mode:

```
# Run Pip's image inside of a new container
$ docker run -dit --name pip pip bash

# Run a command in the running container
$ docker exec pip whoami
```

When you are done with the container, you can stop it and remove it:

```
# Stop the container
$ docker stop pip

# Remove the container
$ docker rm pip
```

Before removing the container, make sure that you have saved all your changes: any unsaved changes will be lost.

Your development environment is ready.

1.4 Step-by-step installation

This section describes step-by-step how to get a development environment on your host machine. We assume that your machine is running a Debian-based Linux distribution.

1.4.1 Installing the required packages

Update the apt package index:

```
$ sudo apt update
```

For the x86 architecture, install the following necessary packages:

```
$ sudo apt install build-essential doxygen gdb git grub2-
common grub-pc haskell-stack nasm opam qemu-system-i386
texlive texlive-latex-extra xorriso
```

For the ARMv7 architecture, install the following necessary packages:

```
$ sudo apt install build-essential doxygen gcc-arm-none-
eabi gdb-multiarch git grub2-common grub-pc haskell-stack
opam qemu-system-arm texlive texlive-latex-extra xorriso
```

Download the GHC compiler if necessary in the `$HOME/.stack`:

```
$ stack setup
```

Initialize the internal state of opam in the `$HOME/.opam` directory:

```
$ opam init
$ eval $(opam env)
```

Build Coq from source with `opam`:

```
$ opam pin add coq 8.13.1
```

1.4.2 Getting source code

First, you have to clone the `pipcore` repository which contains the kernel, proof and documentation of Pip:

```
$ git clone https://github.com/2xs/pipcore.git
```

Then, you may need the source code of the userland library of Pip, called LibPip, which provides useful functions for calling the API or managing the data structures of Pip:

```
$ git clone https://github.com/2xs/libpip.git
```

1.4.3 Building LibPip

To build a partition on top of Pip, you will probably need LibPip.

To build Libpip for the x86 architecture:

```
$ make -C /path/to/libpip ARCH=x86
```

To build LibPip for the ARMv7 architecture:

```
$ make -C /path/to/libpip ARCH=armv7
```

1.4.4 Building Digger

In order to convert the Coq code into C code, you need to build the extractor, called Digger. The first step is to download the source code:

```
# Initialize your local configuration file
$ git -C /path/to/pipcore submodule init

# Fetch all the data from the digger project
$ git -C /path/to/pipcore submodule update
```

Then, build Digger through the stack tool:

```
$ make -C /path/to/pipcore/tools/digger
```

1.4.5 Configuration script

The purpose of the configuration script is to detect whether the tools needed to compile the project are installed. This script expects three mandatory arguments: the target architecture, the name of the root partition and the path to the LibPip. Optional arguments can also be provided. For more information:

```
./path/to/pipcore/configure.sh --help
```

To configure the project for the x86 architecture and the minimal root partition:

```
./path/to/pipcore/configure.sh \
    --architecture=x86 \
    --partition-name=minimal \
    --libpip=/path/to/libpip
```

To configure the project for the ARMv7 architecture and the minimal root partition:

```
./path/to/pipcore/configure.sh \
    --architecture=armv7 \
    --partition-name=minimal \
    --libpip=/path/to/libpip
```

Your development environment is ready.

2 Testing your development environment

This section describes how to test your development environment, whether it is from a virtual machine image, a Docker image or your host machine.

2.1 Building pipcore

You can build pipcore with the root partition on top of it:

```
    $ make -C /path/to/pipcore
```

You should find in `/path/to/pipcore` directory the ELF binary and the ISO image of Pip.

2.2 Testing in QEMU

You can test the ELF version of Pip in QEMU:

```
    $ make -C /path/to/pipcore qemu-elf
```

or test the ISO version:

```
    $ make -C /path/to/pipcore qemu-iso
```

This should display "Hello world!" on the serial link after a few seconds.

3 User Guide

3.1 The minimal partition

The purpose of the minimal partition is to show how to make a functional minimal partition that prints "Hello World" on the serial link without the LibPip. To go into details, see the source code of the minimal partition.

3.1.1 Calling the API of Pip

In order to keep the minimal partition as minimal as possible, we will not use the LibPip library, but rather call the Pip API directly using inline assembly.

Before writing a character on the serial link, it is necessary to check if it is ready to transmit. We must therefore write a function that allows us to retrieve the state of the transmitting cycle of the serial link contained in the Line Status Register (LSR). This register is accessible in read mode at address `0x3FD` (`SERIAL_PORT+5`). Since we are in *userland*, we cannot directly read the IO port using the `IN` instruction. We will have to call the

corresponding Pip service which is located at index `0x38` in the Global Descriptor Table (GDT).

The function of the minimal partition that call the `IN` service of Pip to retrieves the state of the transmitting cycle is the following:

```
uint32_t serial_transmit_ready(void) {
    register uint32_t result asm("eax");
    asm (
       "push %1;"
       "lcall $0x38,$0x0;"
       "add $0x4, %%esp;"
       /* Outputs */
       : "=r" (result)
       /* Inputs */
       : "i" (SERIAL_PORT+5)
       /* Clobbers */
       :
    );
    return result & 0x20;
}
```

This Pip service expects to return the value read on the IO port in the `EAX` register of the CPU. We therefore declare a variable that will be stored in this register:

```
register uint32_t result asm("eax");
```

It also expects to have one argument on the stack, which is the address of the IO port to read. So we push on the stack the argument `%1`, which is the `SERIAL_PORT+5` argument present as input operand:

```
push %1;
```

Now that we have a variable stored in the `EAX` register and pushed the argument onto the stack, we can make our far call:

```
lcall $0x38,$0x0;
```

We clear the stack after the far call by adding `4` to the `ESP` register:

```
add $0x4, %%esp;
```

We define as output operand our `result` variable which will contain the state of the LSR after the far call. `"=r"` is an operand constraint where `"="` means that it is an output operand and `"r"` means that the operand is a register:

```
/* Outputs */
: "=r" (result)
```

We define as input operand the value `SERIAL_PORT+5` which is the address of the IO port to read. `"i"` means that it is an immediate value:

```
/* Inputs */
: "i" (SERIAL_PORT+5)
```

Since we have not clobbers any registers other than the output register, we can provide an empty list:

```
/* Clobbers */
:
```

We return `result & 0x20` because the state of the transmitting cycle is set on bit `5` of the LSR:

```
return result & 0x20;
```

So this function returns `0` if the serial link is not ready or a value other than `0` otherwise.

Now that we have a function to check if the serial link is ready to transmit, we can write a function to print a character. In order to print a character on the serial link, we must write to address `0x3F8` (`SERIAL_PORT`). As we are still in *userland*, we cannot write directly to the IO port using the `OUT` instruction. We will have to use the corresponding Pip service which is located at index `0x30` in the GDT.

The function of the minimal partition that call the `OUT` service of Pip to prints a character on the serial link is the following:

```
void serial_putc(char c) {
    asm (
        "push %1;"
        "push %0;"
        "lcall $0x30, $0x0;"
        "add $0x8, %%esp"
        /* Outputs */
        :
        /* Inputs */
        : "i" (SERIAL_PORT),
          "r" ((uint32_t) c)
        /* Clobbers */
        :
    );
}
```

Now that we have these two functions, we can write our `serial_puts` which writes a string to the serial link:

```
void serial_puts(const char *str) {
    for (char *it = str; *it; ++it) {
        while(!serial_transmit_ready());
        serial_putc(*it);
    }
}
```

Finally, we can print our "Hello World" on the serial link using our `serial_puts` function:

```
void _main()
{
    const char *Hello_world_str = "Hello World !\n";
    serial_puts(Hello_world_str);
    for(;;);
}
```

3.1.2 Linker script

The linker script is use to specify the format and layout of the final executable.

We start by defining the output format, which is always a flat binary, and then the entry point of the partition which is `_main`:

```
OUTPUT_FORMAT(binary)
ENTRY(_main)
```

We define the mandatory `.text` section at address `0x700000`, a `.data` section for the `.data` and `.rodata` and a `.bss` section for the `.bss`:

```
SECTIONS {
    .text 0x700000 :
    {
        *(.text)
        . = ALIGN(0x1000);
    }

    .data :
    {
        *(.data)
        *(.rodata)
        . = ALIGN(0x1000);
    }

    .bss :
    {
        *(.bss)
    }
    end = .;
}
```

The `.text` and `.data` sections are aligned to the size of a page using `ALIGN(0x1000)`.

3.1.3 Makefile

The Makefile is a file allowing to describe the steps necessary to the generation of executables.

We start by declaring the `CFLAGS` which contains the flags used to compile the minimal partition into intermediate objects:

```
CFLAGS=-m32 -c -nostdlib --freestanding -fno-stack-protector
    -fno-pic -no-pie
```

The meaning of the flags is:

`-m32` Generate code for a 32-bit environment.

`-c` Do not use the linker.

`-nostdlib` Do not use the standard system startup files or libraries when linking.

`--freestanding` Do not assume that standard functions have their usual definition.

`-fno-stack-protector` Disable the stack protection.

`-fno-pic` Disable the generation of position-independent code.

`-no-pie` Disable the generation of position independent executable.

We then declare the `LDFLAGS` which contains the flags used to link the minimal partition executable:

```
LDFLAGS=-m elf_i386 -T link.ld
```

The meaning of the flags is:

`-m elf_i386` Create an executable that can run on `elf_i386` processor.

`-T link.ld` Use the linker script that we declare in the previous section.

Finally, we define some generic rules for our sources, and invoke the required compiler for each one, calling the linker once everything has been done:

```
CSOURCES=$(wildcard *.c)

COBJ=$(CSOURCES:.c=.o)

EXEC=minimal.bin

all: $(EXEC)
	@echo Done.

clean:
	rm -f $(COBJ) $(EXEC)

$(EXEC): $(COBJ)
	$(LD) $^ -o $@ $(LDFLAGS)

%.o: %.c
	$(CC) $(CFLAGS) $< -o $@
```

3.2 The launcher partition

The purpose of the launcher partition is to show how a parent partition creates and transfers its execution flow to a child partition. To go into details, see the source code of the launcher partition, which can be downloaded on the Pip Protokernel website.

3.2.1 Creating a child partition

The first step to create a child partition is to allocate five memory pages, using the `Pip_AllocPage` function, for the data structures `descChild`, `pdChild`, `shadow1Child`, `shadow2Child` and `configPagesList`:

```
uint32_t descChild        = Pip_AllocPage();
uint32_t pdChild          = Pip_AllocPage();
uint32_t shadow1Child     = Pip_AllocPage();
uint32_t shadow2Child     = Pip_AllocPage();
uint32_t configPagesList = Pip_AllocPage();
```

For more information about these data structure, please read the PipInternals.md file.

We ask Pip to create a child partition using the `Pip_CreatePartition` function, providing the previous five memory pages as arguments:

```
Pip_CreatePartition(descChild, pdChild,
    shadow1Child, shadow2Child, configPagesList);
```

Once the child partition has been created, we need to map, using the `Pip_MapPageWrapper` function, each page of the child partition image, starting with the one at the `base` address, into the virtual memory of the newly created partition, starting at the `loadAddress` address:

```
for (uint32_t offset = 0; offset < size; offset += PAGE_SIZE)
{
    map_page_rcode = Pip_MapPageWrapper(base + offset,
    descChild, loadAddress + offset);
    /* Error handling */
}
```

When all pages have been mapped, we need to allocate a memory page for the stack of the child partition:

```
uint32_t stackPage = Pip_AllocPage();
```

It is now necessary to create a context for the child partition. This context must be at the beginning of the stack. Since the stack grows downwards from the top of the memory page, the context must be at the end of the page, at the physical address `stackPage + PAGE_SIZE - sizeof(user_ctx_t)`:

```
user_ctx_t *contextPAddr = (user_ctx_t*) (stackPage +
    PAGE_SIZE - sizeof(user_ctx_t));
```

and at the virtual address `STACK_TOP_VADDR + PAGE_SIZE - sizeof( user_ctx_t)` where `STACK_TOP_VADDR` is the virtual address where the stack will be mapped:

```
user_ctx_t *contextVAddr = (user_ctx_t*) (STACK_TOP_VADDR +
    PAGE_SIZE - sizeof(user_ctx_t));
```

The `user_ctx_t` structure contains the following members:

`valid` This member indicates whether the structure is valid or not.

`eip` This member must point to the first instruction of the child.

`pipflags` This member member indicates whether the structure wants to be in virtual `sti` or in virtual `cli`.

`eflags` This member member indicates the state of the context (it is forced to `0x202`).

`ebp` This member must point to the base address of the stack page.

`esp` This member must point to the top of stack.

We now fill the data structure with the appropriate values:

```
contextPAddr->valid    = 0;
contextPAddr->eip      = loadAddress;
contextPAddr->pipflags = 0;
contextPAddr->eflags   = 0x202;
contextPAddr->regs.ebp = STACK_TOP_VADDR + PAGE_SIZE;
contextPAddr->regs.esp = contextPAddr->regs.ebp - sizeof(
    user_ctx_t);
contextPAddr->valid    = 1;
```

Once the data structure is filled, we need to map the stack of the child partition to the virtual address `STACK_TOP_VADDR`:

```
map_page_rcode = Pip_MapPageWrapper(stackPage, descChild,
    STACK_TOP_VADDR);
/* Error handling */
```

We now need to allocate a new memory page for the virtual Interrupt Descriptor Table (IDT):

```
user_ctx_t **vidtPage = Pip_AllocPage();
```

This table allows the child partition to associate an interrupt with a handler. Here, we register the virtual address of the context of the child partition at address `0`, `48` and `49`:

```
vidtPage[ 0] = contextVAddr;
vidtPage[48] = contextVAddr;
vidtPage[49] = contextVAddr;
```

Finally, we map the virtual IDT memory page to the virtual memory address `VIDT_VADDR`:

```
map_page_rcode = Pip_MapPageWrapper((uint32_t) vidtPage,
    descChild, VIDT_VADDR);
/* Error handling */
```

3.2.2 Yielding to a child partition

To transfer the execution flow from a parent partition to a child partition, we have to use the `Pip_Yield` service. Thus, the transfer of the execution flow from the root partition of the launcher to the child partition looks like:

```
Pip_Yield(descChild, 0, 49, 0, 0);
```

This will save the caller context at index `49`. Then this triggers interrupt `0` in the virtual IDT of the child partition designated by `descChild` and loads the context that was saved at that index, which is the child context.

3.2.3 Handling an interruption

The root partition of the launcher handles two interrupts which are the timer interrupt and the keyboard interrupt.

To handle an interrupt, we need to create an interrupt handler. An interrupt handler is simply a function that will be called if the corresponding interrupt has been triggered. The timer interrupt handler of the root partition looks like:

```
void timerHandler(void)
{
    printf("A timer interruption was triggered ...\n");

    // Yield to the child partition
    doYield();

    // Should never be reached
    PANIC();
}
```

Once we have declared an interrupt handler, we need to allocate a page for the handler stack using the `Pip_AllocPage` service:

```
uint32_t handlerStackAddress = Pip_AllocPage();
```

and an interruption context using the `Pip_AllocContext` service:

```
user_ctx_t *timerHandlerContext = Pip_AllocContext();
```

Now we need to register the level `32` interrupt, which is the timer interrupt, with the timer handler using the `Pip_RegisterInterrupt` service:

```
Pip_RegisterInterrupt(timerHandlerContext, 32, timerHandler,
    handlerStackAddress, 0);
```

3.3 The nanny busy beaver partition

The purpose of this partition is to loosely test most of Pip services. To go into details, see the source code of the nanny busy beaver partition, which can be downloaded on the Pip Protokernel website.

3.3.1 Deleting a child partition

The nanny busy beaver partition is similar to the launcher partition in that it creates and transfers its execution flow to a child partition. The only difference is that the partition shows how a parent partition deletes a child partition.

Before deleting a child partition, the parent partition must remove the memory pages given to the child partition. To do this, we must call the `Pip_RemoveVAddr` service:

```
Pip_RemoveVAddr(descChild, removableVPage);
```

This service takes as argument the partition descriptor of the partition and the address of the memory page to remove.

Once the memory pages are removed, we must ask the kernel to collect the removed memory pages. To do this, we have to use the `Pip_Collect` service:

```
Pip_Collect(descChild, removedVPage);
```

This service takes as argument the partition descriptor of the partition and the address of the removed memory page.

Finally, when all the memory pages have been recovered by the parent partition, we can delete the child partition using the `Pip_DeletePartition` service:

```
Pip_DeletePartition(descChild);
```

This service takes as argument the partition descriptor of the partition to be deleted.