Project 4: Hacking with QEMU and KVM

Objective

In order to get in-depth knowledge about full system emulation and hardware virtualization, in this project, students are expected to make small changes in QEMU and KVM to implement a small functionality. In particular, we aim to intercept each executed instruction and print out brief information about each instruction, in both settings: full system emulation and hardware virtualization.

Preparation

1. Boot from live cd
2. Download qemu/kvm source code from http://amun.ecs.syr.edu/~hyin/TC2010/kvm-88.tar.gz
3. Go to the directory where kvm-88 tarball is downloaded and unzip it:
   tar xzf kvm-88.tar.gz
4. Now the source code is extracted into kvm-88 directory:
   cd kvm-88
5. Configure and build from source code:
   ./configure
   make
   sudo make install
6. Install kvm kernel modules:
   sudo modprobe kvm
   sudo modprobe kvm-intel
   sudo chmod 666 /dev/kvm
7. Get a small virtual machine image from
   http://amun.ecs.syr.edu/~hyin/TC2010/small.ffs.bz2
8. Unzip the image
   tar xjf small.ffs.bz2
9. Run qemu with hardware virtualization:
   x86_64-softmmu/qemu-system-x86_64 <path-to>/small.ffs -monitor stdio
10. Run qemu with full system emulation:
    x86_64-softmmu/qemu-system-x86_64 <path-to>/small.ffs -monitor stdio -no-kvm

Trace instructions in full system emulation

The key is to understand the work flow of dynamic binary translation in QEMU. An important file to look into is located in target-i386/translate.c, which takes care of dynamic binary translation. An important function in this file needs attention is disas_insn, which disassemble each instruction and generate target code and put the target code into the translated code cache.

Therefore, we need to insert a function in the beginning of disas_insn to trace each instruction. Note that directly inserting a function like trace_insn(…) is not going to work, because this function is called in the translation time. What we really need is to insert a function call into the translated code, so when each instruction is executed at runtime, our inserted function will be executed.

To define a new function and insert it into the generated code, we need to do the following things:

1. To define your own function, in target-i386/op_helper.c, add something like below:
   void helper_trace_insn(uint32_t eip)
   {
   //whatever you want to print
   }
2. To declare this function as an inserted function, in target-i386/helper.h, add the following:
   DEF_HELPER_1(trace_insn, void, i32)
3. To insert this function, in the beginning of disas_insn function, add:
   gen_helper_trace_insn(pc_start);

Trace instruction in hardware virtualization

To trace instruction in hardware virtualization, we have to use hardware debugging features like single step mode. Basically, by setting the trap flag in CPU eflags, the CPU will raise a single-step exception for each instruction.

QEMU provides a monitor command “singlestep” to enable or disable single step, which is implemented in do_singlestep() function in monitor.c. However, this implementation only works for emulation mode. So we need to change this function to update the cpu state in KVM. We can use cpu_single_step() function to achieve this purpose. For example, we can call “cpu_single_step(first_cpu, 1) ; “ to enable single step and “cpu_single_step(first_cpu, 0);” to disable it.

Now, for each instruction, a single-step exception will be raised within the virtual machine and will be trapped into KVM. Then KVM will hand over this exception to QEMU. More specifically, please look at the kvm_run function in qemu-kvm.c. When KVM hands over the exception to QEMU, it will examine the exit reason in a switch-case statement. For single-step exception, it will fall into the case KVM_EXIT_DEBUG and then will be handled in handle_debug() function. So in handle_debug() function, we can add our own code to print information about the current instruction to handle this case.

Note that after each instruction, the TF flag for single step will be cleared by CPU automatically. So to trace instruction continuously, we need to make sure TF/RF is set after each instruction. We can call kvm_update_guest_debug() function to update the flags.

Note that since we only need to change source code in QEMU, we do not need to reload kvm kernel module with these changes.