Information systems security

Lorenzo Di Maio

October 2024

Contents

Chapter 1

First lab

1.1 User Mode Emulation

Function	Command in Listings
`arm-linux-gnueabi-gcc -o <filename.o> -static <filename.c>`backgroundcolor=	
Font style	`basicstyle=`
Line breaks	`breaklines=true`
Column formatting	`columns=fullflexible`
Line numbers	`numbers=left`
Line number style	`numberstyle=`
Comment style	`commentstyle=`
Keyword style	`keywordstyle=`
String style	`stringstyle=`
Inline comment style	`moredelim=**[is][]////`
Frame around code	`frame=single`

Table 1.1: Listings Command Functionality Table

2. Compile the Hello World Program for ARM

Compile the program for ARM using the GNU ARM toolchain:

```
arm-linux-gnueabi-gcc -o <filename.o> -static <filename.c>
```

3. Attempt to Run the ARM Executable on the Host Machine

Run the ARM binary on the host machine:

```
./helloarm.o
```

This will produce the following error:

```
-bash: ./helloarm.o: cannot execute binary file: Exec format error
```

4. Verify the Architecture of the ARM Executable

Verify the architecture of the ARM binary:

```
file helloarm.o
```

The output should look like:

```
helloarm.o: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, not stripped
```

5. Compile the Hello World Program for the Host Machine

Compile the program for the host machine:

```
gcc -o hellohost.o hello.c
```

6. Run the Program Compiled for the Host Machine

Run the host machine binary:

```
./hellohost.o
```

You should see:

```
Hello, World!
```

7. Verify the Architecture of the Host Executable

Verify the architecture of the host binary:

```
file hellohost.o
```

The output will indicate the architecture, for example:

```
hellohost.o:  ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, not stripped
```

8. Install QEMU for User Mode Emulation

Install QEMU for user mode emulation:

```
sudo apt install qemu-user
```

9. Run the ARM Program on QEMU in User Mode Emulation

Run the ARM binary on your host machine using QEMU:

```
qemu-arm helloarm.o
```

You should see:

```
Hello, World!
```

1.1.1 Conclusion

This exercise demonstrates how to compile and run a program for a different architecture using QEMU's User Mode Emulation.

1.2 Exercise 2: Bare Metal Simulation

Package Installation

Installing QEMU System Emulation

To get started, install the QEMU System Emulation package by running the following command:

```
sudo apt install qemu-system-arm
```

Installing the ARM Cross-Compiler

After installing QEMU System Emulation, you need to install the ARM cross-compiler. For this exercise, we will use the `gcc-arm-none-eabi` package to compile code for a bare metal system. To install it, run the following command:

```
sudo apt install gcc-arm-none-eabi
```

Create the Bare Metal Program

We will create three files: `main.c`, `startup.s`, and `linker.ld`.

Create `main.c` File

The `main.c` file contains a simple program to print "Hello world!" to the UART peripheral. The UART peripheral is memory-mapped to the address `0x40011004`. The program writes each character of the message to the UART peripheral until the end of the string is reached. The program then enters an infinite loop.

Here is the content of the `main.c` file:

```c
volatile unsigned int *const USART1_PTR = (unsigned int *)0x40011004;

void my_printf(const char *s) {
    while(*s != '\0') {  /* Loop until end of string */
        *USART1_PTR = (unsigned int)(*s);  /* Transmit char */
        s++;  /* Next char */
    }
}

int main(void) {
    my_printf("Hello world!\n");
}
```

Create `startup.s` File

The `startup.s` file defines the entry point of the program. This is a simple assembly file where the execution starts.

Here is the content of the `startup.s` file:

```
.word stack_top    // Address of the stack_top
.word _start       // Address of the _start label

.thumb_func
.global _start

_start:
    BL main
    B .
```

Create `linker.ld` File

The `linker.ld` file defines the memory regions of the program, such as the text section (code), data section (initialized variables), bss section (uninitialized variables), and stack section.

Here is the content of the `linker.ld` file:

```
ENTRY(_start)

SECTIONS
{
    .text : { *(.text*) }
    .data : { *(.data) }
    .bss : { *(.bss*) }
    stack_top = 0x2001ffff;
}
```

Compile the Bare Metal Program

Now we need to compile the program.

Compile `startup.s` File

Use the ARM assembler to compile the `startup.s` file into an object file:

```
arm-none-eabi-as -mcpu=cortex-m4 startup.s -o startup.o
```

Compile `main.c` File

Use the ARM GCC compiler to compile the `main.c` file:

```
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb main.c -o main.o
```

Link Object Files to Create `main.elf`

Link the object files (`startup.o` and `main.o`) together using the ARM linker, along with the linker script:

```
arm-none-eabi-ld -T linker.ld startup.o main.o -o main.elf
```

QEMU Simulation

Now that we have the `main.elf` executable, we can run the program using QEMU. This simulates the bare metal ARM system on QEMU with the Netduino Plus 2 board.

To run the program on QEMU, use the following command:

```
qemu-system-arm -M netduino2 -nographic -kernel main.elf
```

Here's what each flag does:

- `-M netduino2`: Specifies the machine type to simulate (in this case, the Netduino Plus 2 board).
- `-nographic`: Runs QEMU without a graphical interface (all output will be in the terminal).
- `-kernel main.elf`: Specifies the kernel (or in this case, the compiled bare metal program) to run.

Ending the Simulation

To stop the simulation in QEMU, press:

```
Ctrl + A, then X
```