



WAYNE STATE UNIVERSITY

COLLEGE OF ENGINEERING

Department of Computer Science: Cyber Security Practice

Lab 6: OS Security for the Internet of Things

Introduction

The Internet of Things (IoT) is an emerging technology that will affect our daily life. It is reported that there would be 100 billion connected IoT devices by 2025, so the impact of IoT on will be impressive and security is an important part. For the purpose of this lab, we will focus on the operating system security for the IoT devices.

There are a number of newly developed operating systems for the IoT. For instance, Contiki is an open source operating system for the Internet of Things. Contiki connects tiny low-cost, low-power microcontrollers to the Internet. In May 2015, Google announced Brillo, an operating system for the IoT. Brillo is a solution from Google for building connected devices, and it is developed based on the Android system. Zephyr is another real time operating system that is designed for IoT devices. Zephyr open source project is announced by Linux foundation in February 2016. In this lab, we use Zehpyr as a study example to explore the OS security of IoT devices. Specifically, we will exploit buffer overflow vulnerabilities in an application and understand the security features of Zephyr OS. After you finish the lab assignment, you will be expected to answer following questions:

- What security features does Zephyr have?
- Do applications share the same address space with the OS kernel?
- Does Zephyr have defense mechanisms such as non-executable stack or Address Space Layout Randomization (ASLR)?
- Do textbook attacks (e.g., buffer overflow or heap spray) work on Zephyr?

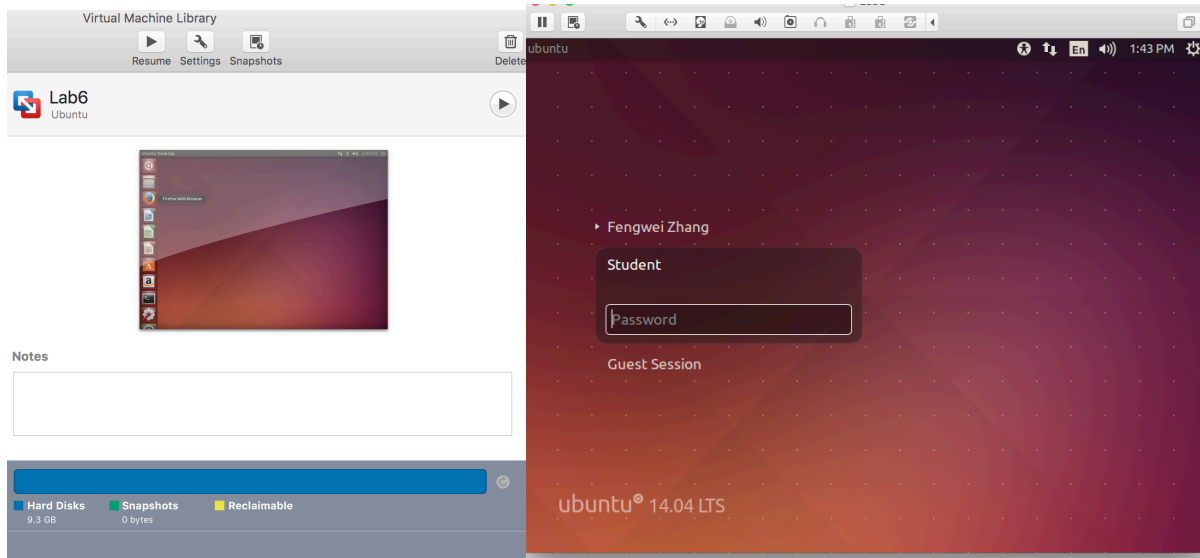
Software Requirements

All required files are packed and configured in the provided virtual machine image.

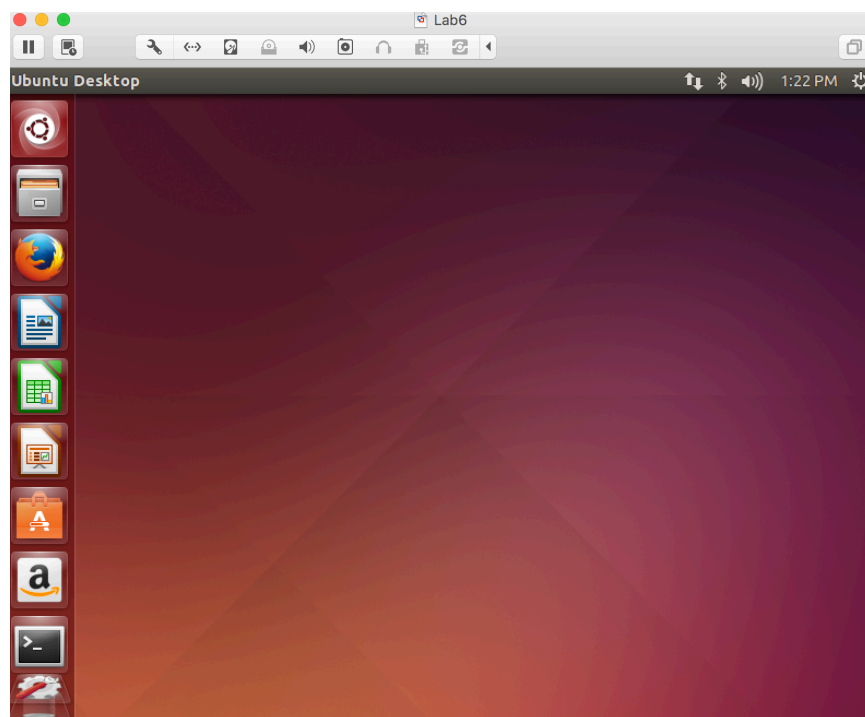
- The VMWare Software
<http://apps.eng.wayne.edu/MPStudents/Dreamspark.aspx>
- The Ubuntu 14.04 Long Term Support (LTS) Version
<http://www.ubuntu.com/download/desktop>
- Zephyr: Real Time OS for IoT – A Linux Foundation Collaborative Project
<https://www.zephyrproject.org/downloads>

Starting the Lab 6 Virtual Machine

In this lab, we use Ubuntu as our VM image. Select the VM named “Lab6.”



Login the Ubuntu image with username student, and password [TBA in the class]. Below is the screen snapshot after login.





Setting up the Zephyr Development Environment

You can find detailed documents from Zephyr Project website:

<https://www.zephyrproject.org/doc>

Download the Zephyr Source Code

The code is hosted at the Linux Foundation with a Gerrit backend that supports anonymous cloning via git.

We can check out the Zephyr source code using git command. You can see that the zephyr-project folder is under the home directory.

<https://www.zephyrproject.org/>

Note that you need to install git if you want to try it on your own machine. Note that our lab image has downloaded the Zephyr source code at `~/zephyr-project/`

Installing Requirements and Dependencies

If you use your own laptop or desktop to do the lab, you need to install the dependencies by executing the following command. On our Ubuntu image, I have installed them for you.

```
$ sudo apt-get install git make gcc gcc-multilib g++ g++-multilib
```

Setting the Project's Environment Variables

```
$ cd zephyr-project
```

```
$ source zephyr-env.sh
```

A terminal window screenshot showing the following commands and their output:

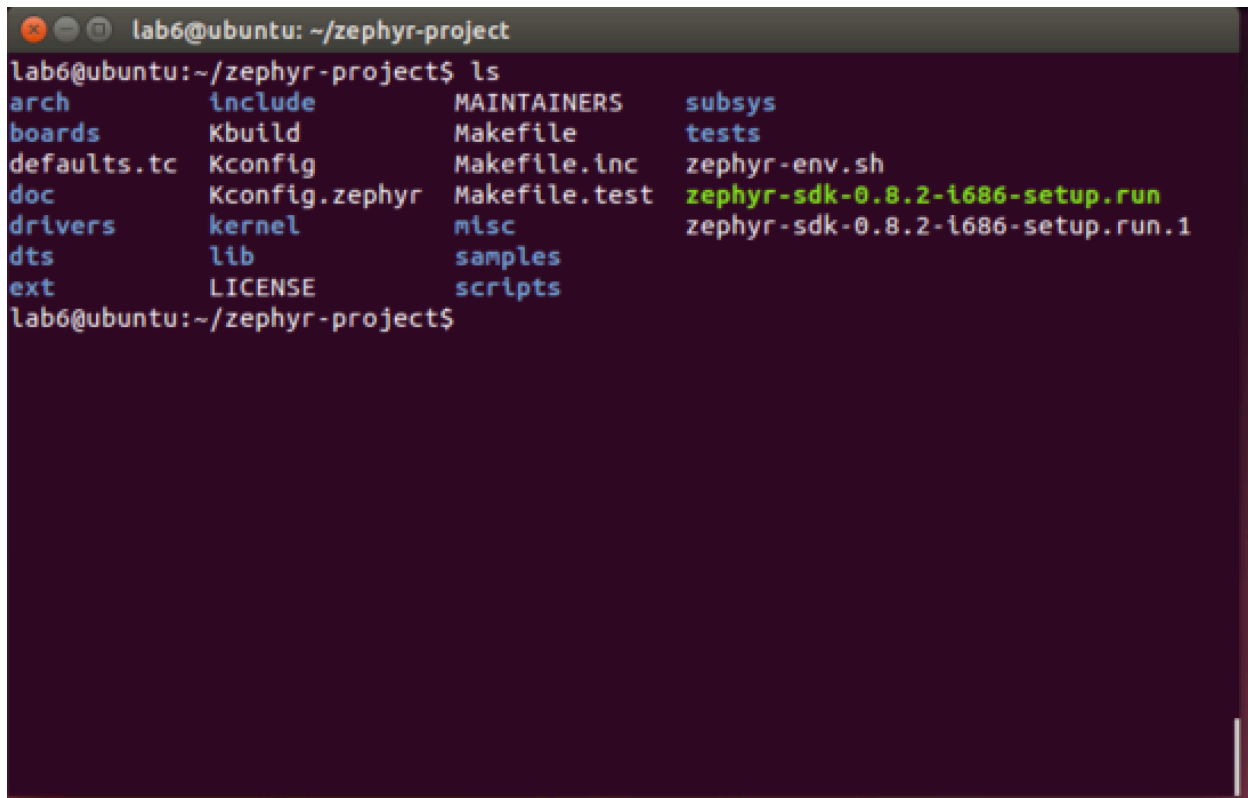
```
lab6@ubuntu: ~/zephyr-project
lab6@ubuntu:~$ cd zephyr-project
lab6@ubuntu:~/zephyr-project$ source zephyr-env.sh
lab6@ubuntu:~/zephyr-project$
```

Installing the Zephyr Software Development Kit

Zephyr's SDK contains all necessary tools and cross-compilers needed to build the kernel on all supported architectures. Additionally, it includes host tools such as a custom QEMU and a host compiler for building host tools if necessary. The SDK supports the following architectures: **IA-32**, **ARM**, and **ARC**.

Next, you need to follow these steps to install the SDK on your Ubuntu Linux VM.

Step 1. Download the SDK self-executable script from zephyr website. The image has downloaded the script; the file name is `zephyr-sdk-0.8.2-i686-setup.run`. See the screenshot below.



```
lab6@ubuntu: ~/zephyr-project
lab6@ubuntu:~/zephyr-project$ ls
arch          include      MAINTAINERS  subsys
boards       Kbuild      Makefile     tests
defaults.tc  Kconfig     Makefile.inc zephyr-env.sh
doc          Kconfig.zephyr Makefile.test zephyr-sdk-0.8.2-i686-setup.run
drivers      kernel      misc         zephyr-sdk-0.8.2-i686-setup.run.1
dts         lib         samples
ext         LICENSE    scripts
lab6@ubuntu:~/zephyr-project$
```



Step 2. Run the installation script

```
$ chmod a+x zephyr-sdk-0.8.2-i686-setup.run
```

```
$ sudo ./zephyr-sdk-0.8.2-i686-setup.run
```

The screenshot below shows the executions of step 1 and 2. We can see that the default directory of Zephyr SDK is installed at /opt/zephyr-sdk directory. Since I have installed the SDK in the image before, you will see a message that ask if you want to remove the existing directory /opt/zephyr. Just select yes.

```
lab6@ubuntu: ~/zephyr-project
lab6@ubuntu:~/zephyr-project$ wget https://nexus.zephyrproject.org/content/repositories/releases/org/zephyrproject/zephyr-sdk/0.8.2-i686/zephyr-sdk-0.8.2-i686-setup.run
--2017-03-14 10:48:25-- https://nexus.zephyrproject.org/content/repositories/releases/org/zephyrproject/zephyr-sdk/0.8.2-i686/zephyr-sdk-0.8.2-i686-setup.run
Resolving nexus.zephyrproject.org (nexus.zephyrproject.org)... 199.19.213.246
Connecting to nexus.zephyrproject.org (nexus.zephyrproject.org)|199.19.213.246|:443... connected
.
HTTP request sent, awaiting response... 200 OK
Length: 390661157 (373M) [application/octet-stream]
Saving to: 'zephyr-sdk-0.8.2-i686-setup.run.1'

100%[=====] 390,661,157 1.61MB/s in 3m 17s

2017-03-14 10:51:42 (1.89 MB/s) - 'zephyr-sdk-0.8.2-i686-setup.run.1' saved [390661157/390661157]

lab6@ubuntu:~/zephyr-project$ chmod a+x zephyr-sdk-0.8.2-i686-setup.run
lab6@ubuntu:~/zephyr-project$ sudo ./zephyr-sdk-0.8.2-i686-setup.run
Verifying archive integrity... All good.
Uncompressing SDK for Zephyr 100%
Enter target directory for SDK (default: /opt/zephyr-sdk/):
Installing SDK to /opt/zephyr-sdk
The directory /opt/zephyr-sdk/sysroots will be removed!
Do you want to continue (y/n)?

Invalid input "", please input 'y' or 'n':
y
[*] Installing x86 tools...
[*] Installing arm tools...
[*] Installing arc tools...
[*] Installing ianctu tools...
[*] Installing mips tools...
[*] Installing nios2 tools...
[*] Installing additional host tools...
Success installing SDK. SDK is ready to be used.
lab6@ubuntu:~/zephyr-project$
```



Step 3. To use the Zephyr SDK, export the following environment variables and use the target location where SDK was installed. You can just add following lines into the `~/.bashrc` file.

```
$ vim ~/.bashrc
```

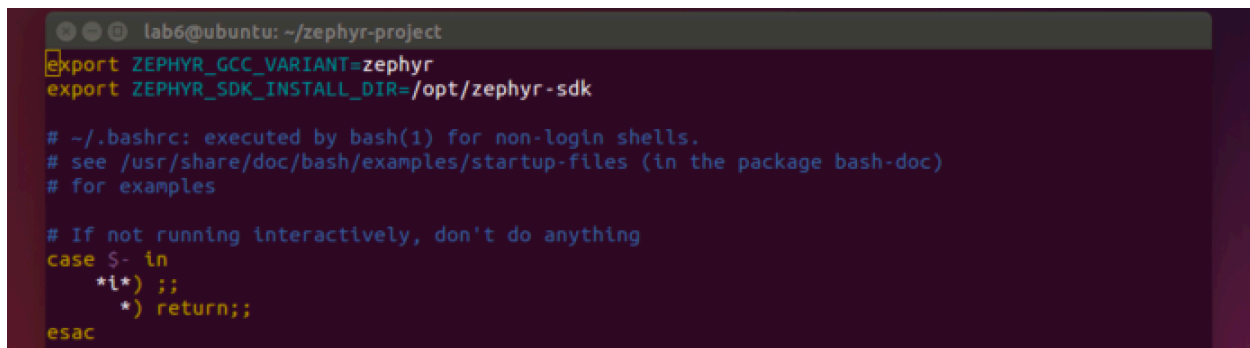
Add these two lines into the file

```
export ZEPHYR_GCC_VARIANT=zephyr
```

```
export ZEPHYR_SDK_INSTALL_DIR=/opt/zephyr-sdk
```

```
$ source ~/.bashrc
```

The screenshot below shows the step 3.

A terminal window screenshot showing the process of editing the `~/.bashrc` file. The prompt is `lab6@ubuntu: ~/zephyr-project`. The user has entered `export ZEPHYR_GCC_VARIANT=zephyr` and `export ZEPHYR_SDK_INSTALL_DIR=/opt/zephyr-sdk`. The terminal also shows the standard `~/.bashrc` header comments and a `case` statement for interactive shells.

```
lab6@ubuntu: ~/zephyr-project
export ZEPHYR_GCC_VARIANT=zephyr
export ZEPHYR_SDK_INSTALL_DIR=/opt/zephyr-sdk

# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
  *(*) ;;
  *) return;;
esac
```

Building and Running an Application with Zephyr

You have successfully setup the development environment for Zephyr. This section provides all the steps to build a Zephyr kernel containing your application and run it. We use the *Hello World* sample application as an example. You can also create your own application and run it.

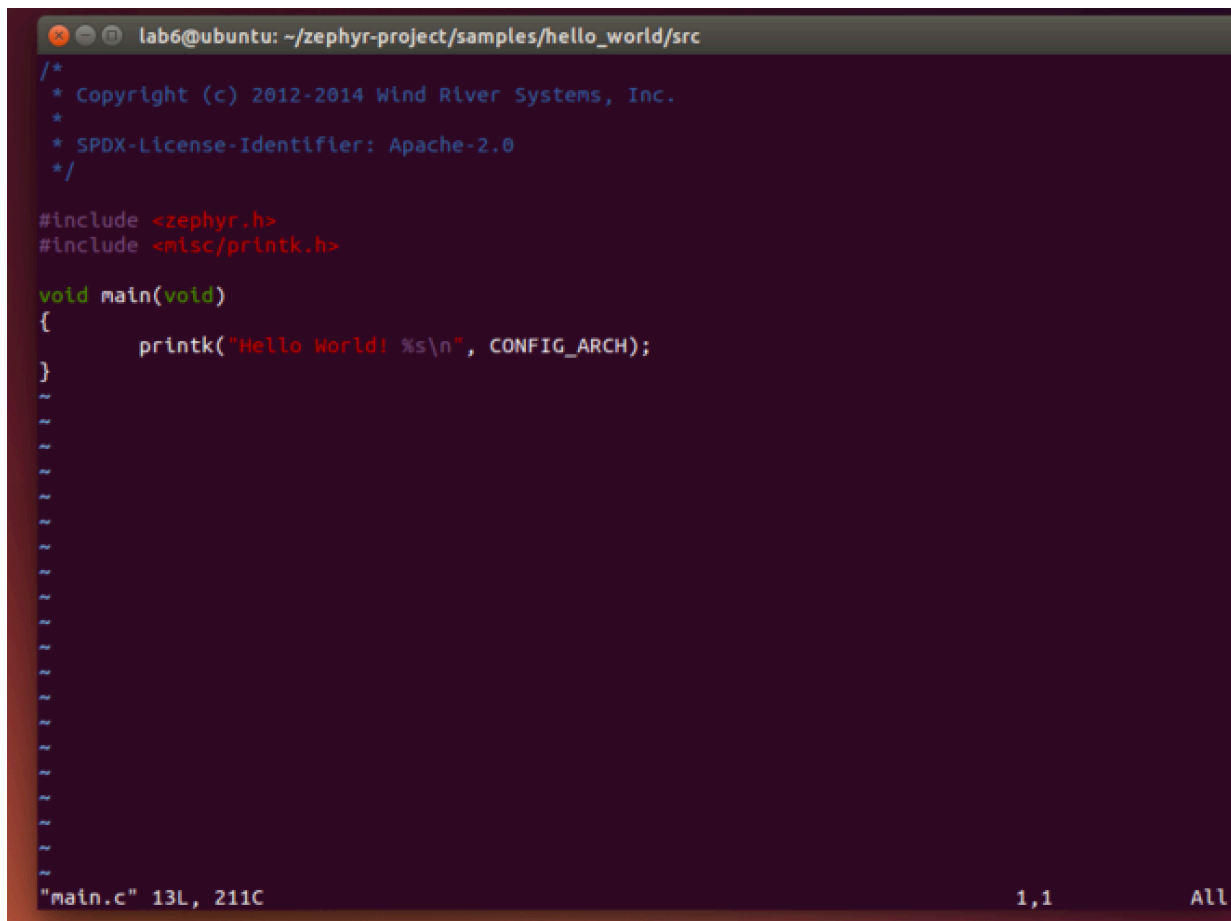
Sample *Hello World* Application

First, let's take a look at what a sample application of Zephyr look like. Go the source directory of the *Hello World* sample.

```
$ cd ~/zephyr-project/samples/hello_world/src
```

```
$ vim main.c
```

The screenshot below shows the source code of the *Hello World* application.



```
lab6@ubuntu: ~/zephyr-project/samples/hello_world/src
/*
 * Copyright (c) 2012-2014 Wind River Systems, Inc.
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr.h>
#include <misc/printk.h>

void main(void)
{
    printk("Hello World! %s\n", CONFIG_ARCH);
}

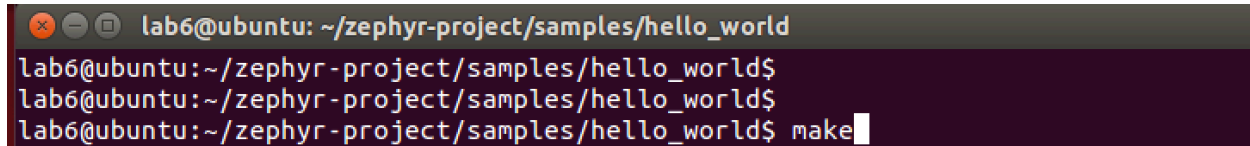
"main.c" 13L, 211C                               1,1           All
```

Building a Sample Application

To build the Hello World sample application, you can just by executing following commands:

```
$ cd ~/zephyr-project/samples/hello_world/
```

```
$ make
```



```
lab6@ubuntu: ~/zephyr-project/samples/hello_world
lab6@ubuntu:~/zephyr-project/samples/hello_world$
lab6@ubuntu:~/zephyr-project/samples/hello_world$
lab6@ubuntu:~/zephyr-project/samples/hello_world$ make
```

The above screenshot of make will build the hello_world sample application using the default settings defined in the application's Makefile. You can build for a different platform by defining the variable BOARD with one of the supported platforms, for example:

```
$ make BOARD=arduino_101
```

The screenshot below shows the result after executing `$ make BOARD=arduino_101`



```
CC      kernel/system_work_q.o
CC      kernel/thread.o
CC      kernel/thread_abort.o
CC      kernel/timer.o
CC      kernel/work_q.o
AR      kernel/lib.a
CC      src/main.o
LD      src/built-in.o
AR      libzephyr.a
LINK    zephyr.lnk
SIDT    staticIdt.o
LINK    zephyr.elf
BIN     zephyr.bin
make[2]: Leaving directory `/home/lab6/zephyr-project/samples/hello_world/outdir/arduino_101'
make[1]: Leaving directory `/home/lab6/zephyr-project'
lab6@ubuntu:~/zephyr-project/samples/hello_world$
```

The screenshot below shows the supported board by Zephyr project including Intel Galileo Gen1 and Gen2. For the purpose of this lab, we will test the application on x86 QEMU. You can also type:

```
$ make help
```

This gets a full list of supported boards and other useful commands.

x86 Instruction Set

- [Arduino 101](#)
- [Quark D2000 CRB](#)
- [Galileo Gen1/Gen2](#)
- [Minnowboard Max](#)
- [X86 Emulation \(QEMU\)](#)

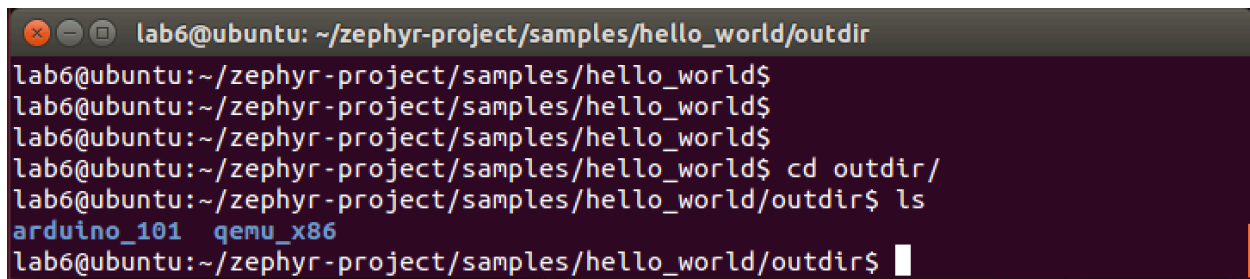
ARM (v7-M and v7E-M) Instruction Set

- [ARM Cortex-M3 Emulation \(QEMU\)](#)
- [Arduino Due](#)
- [Freescale FRDM-K64F](#)

ARC EM4 Instruction Set

- [Arduino 101](#)

The sample projects for the microkernel and the nanokernel are found at `~/zephyr-project/samples` with each sample having a microkernel and nanokernel specific build. After building an application successfully, the results can be found in the `outdir` sub-directory under the application root directory. The ELF binaries generated by the build system are named by default `zephyr.elf`. The screenshot below shows listing the files in the `outdir` directory.



```
lab6@ubuntu: ~/zephyr-project/samples/hello_world/outdir
lab6@ubuntu:~/zephyr-project/samples/hello_world$
lab6@ubuntu:~/zephyr-project/samples/hello_world$
lab6@ubuntu:~/zephyr-project/samples/hello_world$ cd outdir/
lab6@ubuntu:~/zephyr-project/samples/hello_world/outdir$ ls
arduino_101  qemu_x86
lab6@ubuntu:~/zephyr-project/samples/hello_world/outdir$
```

Running a Sample Application

To perform a rapid testing of the *hello world* application, we can use QEMU and the x86 emulation board configuration (`qemu_x86`) by executing the following command:

```
$ cd ~/zephyr-project/samples/hello_world
```

```
$ make BOARD=qemu_x86 qemu
```

```
lab6@ubuntu: ~/zephyr-project/samples/hello_world
lab6@ubuntu:~/zephyr-project/samples/hello_world$
lab6@ubuntu:~/zephyr-project/samples/hello_world$ make BOARD=qemu_x86 qemu
make[1]: Entering directory `/home/lab6/zephyr-project'
make[2]: Entering directory `/home/lab6/zephyr-project/samples/hello_world/outdir/qemu_x86'
  Using /home/lab6/zephyr-project as source for kernel
  GEN      ./Makefile
  CHK      include/generated/version.h
  CHK      misc/generated/configs.c
  CHK      include/generated/offsets.h
  CHK      misc/generated/sysgen/prj.mdef
To exit from QEMU enter: 'CTRL+a, x'
[QEMU] CPU: qemu32
Hello World! x86
```

The screenshot above shows the execution of the *hello world* application using QEMU.

We can see the “Hello World!” is printed out in the terminal. To exit QEMU, we can type “Ctrl+a, x”.

```
To exit from QEMU enter: 'CTRL+a, x'
[QEMU] CPU: qemu32
***** BOOTING ZEPHYR OS v1.7.99 - BUILD: Mar 14 2017 17:56:10 *****
Hello World! x86
QEMU: Terminated
make[2]: Leaving directory `/home/lab6/zephyr-project/samples/hello_world/outdir/qemu_x86'
make[1]: Leaving directory `/home/lab6/zephyr-project'
lab6@ubuntu:~/zephyr-project/samples/hello_world$
```

Exploiting Buffer Overflows in Zephyr Applications

First, let us write a Zephyr application that contains a buffer overflow vulnerability. Change the hello world *main.c* program to the following code as shown in the screenshot.

```
$ gedit main.c
```

```
lab6@ubuntu:~$ cd zephyr-project/samples/hello_world/src/
lab6@ubuntu:~/zephyr-project/samples/hello_world/src$ gedit main.c
lab6@ubuntu:~/zephyr-project/samples/hello_world/src$
```

```
main.c (~/zephyr-project/samples/hello_world/src) - gedit
Open Save Undo Cut Copy Paste Find
main.c x
/*
 * Copyright (c) 2012-2014 Wind River Systems, Inc.
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr.h>
#include <misc/printk.h>

#include <string.h>
void overflow (char *str) {
    char buffer[10];
    strcpy(buffer, str); // Dangerous!
}

int main(void)
{
    char *str = "This is a string that is larger than the buffer size, 10";
    overflow(str);

    return 1;
}
C Tab Width: 8 Ln 2, Col 52 INS
```

From the source code we can see that there is a buffer overflow vulnerability embedded in the *overflow()* function. Then, we pass a long string to the *overflow()* function, the string will overwrite the return address on the stack and the program would crash because of the invalid return address. Next, let's compile the application and run it to see what happens.

As mentioned, to build the application, you can just by executing following commands:

```
$ cd ~/zephyr-project/samples/hello_world
$ make
```

To run the application using QEMU and the x86 emulation board configuration (*qemu_x86*) by executing the following command:

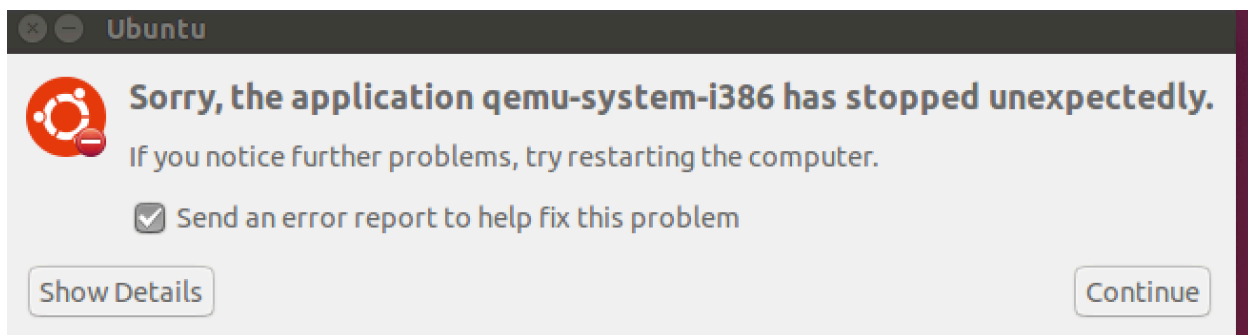
```
$ cd ~/zephyr-project/samples/hello_world
$ make BOARD=qemu_x86 qemu
```

As we expected, the application crashes due to an invalid return address.

```
lab6@ubuntu: ~/zephyr-project/samples/hello_world
CHK include/generated/offsets.h
To exit from QEMU enter: 'CTRL+a, x'
[QEMU] CPU: qemu32
**** BOOTING ZEPHYR OS v1.7.99 - BUILD: Mar 14 2017 17:56:10 ****
qemu: fatal: Trying to execute code outside RAM or ROM at 0x7420676e

EAX=00103156 EBX=00000000 ECX=00101778 EDX=00101740
ESI=00000000 EDI=00000000 EBP=69727473 ESP=00103168
EIP=7420676e EFL=00000246 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
CS =0008 00000000 ffffffff 00cf9b00 DPL=0 CS32 [-RA]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
GS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT= 00100070 00000017
IDT= 00101a30 000007ff
CR0=0000003f CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0fff DR7=00000400
CCS=00000000 CCD=00000000 CCO=LOGICB
EFER=0000000000000000
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000 XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
make[2]: *** [run] Aborted (core dumped)
make[2]: Leaving directory `/home/lab6/zephyr-project/samples/hello_world/outdir/qemu_x86'
make[1]: *** [sub-make] Error 2
make[1]: Leaving directory `/home/lab6/zephyr-project'
make: *** [qemu] Error 2
lab6@ubuntu:~/zephyr-project/samples/hello_world$
```

Furthermore, QEMU also crashes and you will see a pop-up window as the screenshot below.





We can see that the EIP register has the value 0x7420676e. In order to execute something meaningful after exploiting a buffer overflow vulnerability, we need to control the EIP register. Next, adjust the input string in the main.c to change the EIP register to 0x41414141. Note that 0x41 is the ASCII value of character 'A'.

We can just simply edit the main() function as the screenshot below

```
int main (void) {  
    char *str = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";  
    overflow(str);  
  
    return 1;  
}
```

To re-compile the application, you need to re-run the zephyr environment script zephyr-env.sh.

```
$ cd ~/zephyr-project
```

```
$ source ./zephyr-env.sh
```

Then you can re-compile the application and run it. You will see the EIP register will be 0x41414141 as the screenshot below.

```

lab6@ubuntu: ~/zephyr-project/samples/hello_world
qemu: fatal: Trying to execute code outside RAM or ROM at 0x41414141

EAX=00103136 EBX=00000000 ECX=0010176b EDX=00101740
ESI=00000000 EDI=00000000 EBP=41414141 ESP=00103148
EIP=41414141 EFL=00000246 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
CS =0008 00000000 ffffffff 00cf9b00 DPL=0 CS32 [-RA]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
GS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT= 00100070 00000017
IDT= 00101a10 000007ff
CR0=0000003f CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
CCS=00000000 CCD=00000000 CCO=LOGICB
EFER=0000000000000000
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000 XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
make[2]: *** [run] Aborted (core dumped)
make[2]: Leaving directory `/home/lab6/zephyr-project/samples/hello_world/outdir/qemu_x86'
make[1]: *** [sub-make] Error 2
make[1]: Leaving directory `/home/lab6/zephyr-project'
make: *** [qemu] Error 2
lab6@ubuntu:~/zephyr-project/samples/hello_world$

```

```

qemu: fatal: Trying to execute code outside RAM or ROM at 0x41414141

EAX=00104b1e EBX=00000000 ECX=00101f3c EDX=00101f0e
ESI=00000000 EDI=00000000 EBP=41414141 ESP=00104b30
EIP=41414141 EFL=00000246 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]

```

To re-compile the application, you need to re-run the zephyr environment script zephyr-env.sh.

```
$ cd ~/zephyr-project
```

```
$ source ./zephyr-env.sh
```




Application Stack Frame on Zephyr

To do more meaningful things such as executing shell code on the stack, we need to understand the application's stack frame. We have done the similar tasks in the Lab 2. As mentioned, for each Zephyr application, the compilation binaries are stored in a directory called outdir. We can just go to that directory and use objdump tool to disassembly the application binary and understand its stack frame.

```
$ cd ~/zephyr-project/samples/hello_world/outdir/qemu_x86/src
```

```
$ objdump -d main.o
```

The screenshot below shows the disassembly result of main.o binary.

```
lab6@ubuntu: ~/zephyr-project/samples/hello_world/outdir/qemu_x86/src
lab6@ubuntu:~/zephyr-project/samples/hello_world/outdir/qemu_x86$ cd src/
lab6@ubuntu:~/zephyr-project/samples/hello_world/outdir/qemu_x86/src$ ls
built-in.o  main.o
lab6@ubuntu:~/zephyr-project/samples/hello_world/outdir/qemu_x86/src$ objdump -
main.o

main.o:      file format elf32-i386

Disassembly of section .text.__k_mem_pool_quad_block_size_define:

00000000 <__k_mem_pool_quad_block_size_define>:
 0: 55          push   %ebp
 1: 89 e5      mov   %esp,%ebp
 3: 5d        pop   %ebp
 4: c3        ret

Disassembly of section .text.main:

00000000 <main>:
 0: 55          push   %ebp
 1: 89 e5      mov   %esp,%ebp
 3: 68 00 00 00 00  push  $0x0
 8: 68 04 00 00 00  push  $0x4
 d: e8 fc ff ff ff  call  e <main+0xe>
12: 58        pop   %eax
13: 5a        pop   %edx
14: c9        leave
15: c3        ret
lab6@ubuntu:~/zephyr-project/samples/hello_world/outdir/qemu_x86/src$
```



Assignments for Lab 6

1. Read the lab instructions above and finish all the tasks.
2. Answer the questions in the Introduction section, and justify your answers.
Simple yes or no answer will not get any credits.
 - a. What security features does Zephyr have?
 - b. Do applications share the same address space with the OS kernel?
 - c. Does Zephyr have defense mechanisms such as non-executable stack or Address Space Layout Randomization (ASLR)?
 - d. Do textbook attacks (e.g., buffer overflow or heap spray) work on Zephyr?
3. Change the EIP register to the value 0xdeadbeef, and show me the screenshot of the EIP value when the application crashes.

Extra Credit (10pt): Execute shell code on the stack. The shell code could be launching a shell or print a hello string.

Happy Hacking!