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 Zephyr 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

- The Ubuntu 14.04 Long Term Support (LTS) Version

- Zephyr: Real Time OS for IoT – A Linux Foundation Collaborative Project
  [https://www.zephyrproject.org/downloads](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 csc5991-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. You can just open a terminal and run the following command:

$ git clone https://gerrit.zephyrproject.org/r/zephyr zephyr-project

The screenshot above shows checking out the Zephyr source code using git command. You can see that the zephyr-project folder is under the home directory. Note that you need to install git if you want to try it on your own machine.

Update Your Operating System

Before proceeding with the build, ensure your OS is up to date. On our Ubuntu image, execute the following command:

$ sudo apt-get update
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

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.

$ wget URL

https://nexus.zephyrproject.org/content/repositories/releases/org/zephyrproject/zephyr-sdk/0.7.2-i686/zephyr-sdk-0.7.2-i686-setup.run
Step 2. Run the installation script

$ chmod +x zephyr-sdk-0.7.2-i686-setup.run

$ sudo ./zephyr-sdk-0.7.2-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.
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.
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/microkernel/src
```

```
$ vim main.c
```

The screenshot below shows the source code of the *Hello World* application.
Building a Sample Application

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

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

$ 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=minnowboard

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.

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/microkernel/
$ make BOARD=qemu_x86 qemu
```

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”.
Exploiting Buffer Overflows in Zephyr Applications

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

![Image of main.c code]

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/microkernel/

$ 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/microkernel/

$ make BOARD=qemu_x86 qemu

As we expected, the application crashes due to an invalid return address.
Furthermore, QEMU also crashes and you will see a pop-up window as the screenshot below.

![Screenshot of QEMU crash]

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.

```c
int main (void) {
  char *str = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
  overflow(str);

  return 1;
}
```

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

![Screenshot of EIP register change]

To re-compile the application, you need to re-run the zephyr environment script zephyr-evn.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/microkernel/outdir/samples/hello_world/microkernel/src

$ objdump –d main.o

The screenshot below shows the disassembly result of main.o binary.
Assignments for Lab 5

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!