Welcome to the first hands on session of WSE 380 - Honeypots and Intrusion Detection! Today, we are going to start by talking about the Linux command line and finish by setting up our own honeypot. I hope when this session is finished, you will not only know how to setup a honeypot but also have a much better understanding of the Linux command line and how to accomplish tasks using it.

1 Linux Command Line Basics

Before we can setup a honeypot, we will learn some basics of the Linux command line. This will give us the knowledge necessary to install and setup our honeypot, and also give us insight into what attackers are doing once we start to analyze honeypot data.

1.1 The Current Directory

When using any command line environment, it is up to the user to keep a mental map of where in the file system they are currently located. Luckily, there are two easy commands to help us figure out where we are and what files and sub directories are currently around us.

- **pwd** - Lists the absolute path of the current directory

```
user@wse380-22fa:~$ pwd
/home/user/
```

- **ls** - Lists the contents of a directory. Specifying no path will display the contents of the current directory.
  - `ls -l`: Displays additional information about each file such as permissions and file size
  - `ls -a`: Displays all files in a directory, including hidden ones

```
user@wse380-22fa:~$ ls -l -a
total 24
drwxr-xr-x 2 wse380 wse380 4096 Mar 11 13:00 .
drwxr-xr-x 9 root   root   4096 Mar 11 13:00 ..
-rw------- 1 wse380 wse380    5 Mar 11 13:00 .bash_history
-rw-r--r-- 1 wse380 wse380  220 Mar 11 13:00 .bash_logout
-rw-r--r-- 1 wse380 wse380 3771 Mar 11 13:00 .bashrc
-rw-r--r-- 1 wse380 wse380  807 Mar 11 13:00 .profile
```

Above, we used `ls` to list the contents of the current directory while also specifying the `-l` and `-a` flags to show us all files as well as details on each file. Here, from left to right on each row, we can see the permissions on each file and directory, the user that owns it, the user group that has access to it, its size in bytes, the date and time it was last modified, and its name.

1.2 Moving Around the File System

One of the most important symbols in the Linux command line is the ".", or period symbol. In the context of the file system, "./" represents the directory the user is currently in, and "../" represents the parent directory of the current directory. For the most part, when attempting to navigate the file system or interact with files, you will use file paths relative to your current directory. These are called *relative paths*. For example, if a user was trying to list the contents of the current directory's parent, they would use the following command:

```
user@wse380-22fa:~$ ls ../
```

The notion of relative file paths becomes vital when attempting to navigate the file system. To accomplish this, we make use of the "change directory" or `cd` command.

- **cd** - Change the current directory to the one specified. Supplying no arguments navigates to the user's home directory.

```
user@wse380-22fa:~$ ls
wse380Lessons/
user@wse380-22fa:~$ cd wse380Lessons/
user@wse380-22fa:~/wse380Lessons$ pwd
/home/user/wse380Lessons
user@wse380-22fa:~/wse380Lessons$ cd ../
```

Above, we issued a series of commands to help visualize our movement around the file system. We first used ls to view the files and sub directories we had available to us from our current directory. We then used the `cd` command to move to the "wse380Lessons" sub directory. A call to the `pwd` command shows that our new current directory was now the Desktop located at "/home/user/wse380Lessons". Finally, we used the `cd` command with the relative path "../" to move back to the parent directory, the original directory we started at.

An example of an *absolute path* would be the output of the earlier `pwd` command: "/home/user/wse380Lessons". We could have, for example, executed `cd /home/user/wse380Lessons` from anywhere in the system instead of needing to execute `cd wse380Lessons` from our home directory. This is powerful because absolute paths enable us to *refer to any part of the file system, regardless of our current working directory*.

1.3 Creating, Reading, Moving, Copying, and Deleting Files and Directories

- **touch** - Create a file with the specified name.

  ```
  user@wse380-22fa:~$ touch example.txt
  ```

- **cat** - Print a file's contents to the terminal window.

  ```
  user@wse380-22fa:~$ cat example.txt
  ```

- **mkdir** - Create a new directory with the specified name.

  ```
  user@wse380-22fa:~$ mkdir exampleFiles
  ```

- **mv** - Move a file from one location in the file system to another. Also the method used to rename files from the command line.

  ```
  user@wse380-22fa:~$ mv example.txt exampleFiles/example.txt
  ```

- **cp** - Copy a file from one location in the file system to another while keeping the original file untouched. When copying a directory, the user must supply the "-R" argument.

  ```
  user@wse380-22fa:~$ cp example.txt exampleFiles/example.txt
  ```

- **rm** - Remove a file from the file system. To remove a directory, the user must include the "-r" argument to *recursively* remove all sub directories and files. **IMPORTANT: Removing files with this command is permanent! This command is the equivalent of using your graphical interface to move a file to the trash and emptying it. Only remove a file or directory you are sure you want permanently deleted!**

  ```
  user@wse380-22fa:~$ rm -r exampleFiles/
  ```

1.4 Editing Files

You may be used to writing code and editing configuration files using an IDE. While many of the most popular IDEs are available on Linux, it is important to know how to edit files from the Linux command line since you will not be able to use a graphical text editor or IDE to edit files on remote servers.

The most popular command line text editors on Linux include: Vim, Nano, and Emacs. We will be using Nano in this course as it has the easiest learning curve. To get started, run the following command to open a file with Nano:

```
user@wse380-22fa:~$ nano example.txt
```

Here, *example.txt* can either be a new file or an existing file. You will then be able to enter any new text or edit any existing text in the file. Once you are ready to save your work, press *Ctrl+x* to exit Nano. You will be asked if you'd like to save the changes that you made. If so, press *Y*, otherwise, press *N*. Lastly, you will be asked to provide the name of the file you'd like to write your changes to. By default, this will be the name of the file you entered when opening Nano. Most of the time we'd just press enter here. However, if you'd like to keep the original file the same as when you opened it and write your changes to a new file, enter the name of the new file and press enter.

1.5 Downloading Programs

When you downloaded and installed applications on your computer in the past, you most likely did so either through an application marketplace or by downloading them from a vendor's website and running an installer program. On Linux, you can download and install programs using the command line program *apt* or "Advanced Package Tool". In many ways, *apt* is easier to use than the process you are familiar with. The only command we have to learn is:

```
user@wse380-22fa:~$ sudo apt install program_name
```

Here, "program_name" is the name of the program you would like to install. In order to install applications using *apt*, we must run it with administrator privileges. We will discuss this in Section 1.8.

1.6 Downloading Code Using Git

As you may be familiar with from your own programming assignments, `git` allows us to upload our code to the cloud for sharing and collaboration. For this course, we are only interested in how to download code from a remote git repository to our machine. To do this we enter the following command:

```
user@wse380-22fa:~$ git clone https://github.com/...
```

Let us clone the repository with the material for this semester by running:

```
user@wse380-22fa:~$ git clone https://github.com/briankon116/wse380Lessons.git
```

1.7 Running Programs

After we download a program or write our own, we want to actually run them. Doing so is easy, although the syntax of the command differs depending on what kind of program it is. If a program was downloaded using *apt*, it can be executed simply by typing the name of the program into the terminal. For example, if we downloaded the Internet browser Firefox using *apt* we could run it like so:

```
user@wse380-22fa:~$ firefox
```

However, if we wrote our own program or downloaded a program from another source like GitHub, we need to address the executable file directly in the terminal using the same relative paths we talked about earlier. For example, if we downloaded a GitHub repository containing an executable file named *foo*, we could run it by first navigating into the directory containing *foo*, and executing the command:

```
user@wse380-22fa:~$ ./foo
```

When executing any kind of program, there will most likely be data outputted to the terminal window. There are two standard places where a program writes its output: standard out (`stdout`) for regular program output and standard error (`stderr`) for error and log messages. Sometimes we are interested in saving this output to a file for later processing. To do so, we will use the ">" symbol to tell Linux to redirect the `stdout` of the command to a file of our choosing. Take for instance the following command:

```
user@wse380-22fa:~$ ./foo > foo_output.log
```

Here, instead of `stdout` of `foo` going to the terminal window, it will now appear in a new file called `foo_output.log` in the current working directory. If we are also interested in the `stderr` of the program, we can run it as follows:

```
user@wse380-22fa:~$ ./foo > foo_output.log 2>&1
```

Here, we redirect both `stdout` and `stderr` of `foo` to the file `foo_output.log` in the current working directory.

1.8 Sudo

So far, we learned all of the basic Linux commands we need to setup and manage our honeypots. However, some of the commands we execute with honeypots require administrator, or *root*, privileges. On Linux machines, the root user has permission to do just about anything it wants, even modify or delete critical operating system files. Due to this immense power, we typically don't directly sign in with the root user account, but rather permit regular user to execute commands as root as they see fit by explicitly stating their intent to do so. This prevents users from accidentally performing dangerous commands without thinking.
To state our intent to run a command as root, we prepend the keyword *sudo* before the command we would like to run. For example, if we would like to open a file in Nano with root privileges, we would enter the following command:

```
user@wse380-22fa:~$ sudo nano example.txt
```

You will sometimes be asked to enter your password when you attempt to run a sudo command. Just enter the same password you used to log into your account, and you should be good to go, assuming your user account is permitted to execute sudo commands.

1.9 Tab Completion

Most shells offer *tab completion* — if you begin to type part of a file and press the tab key, the shell should autocomplete the name for you (assuming there is only one file that matches the prefix you have typed). If the prefix matches multiple files, you can press tab multiple times, and your shell should attempt to list all files that match the prefix and cycle through them.

2 Setup and Administration of a Low Interaction Honeypot

Now that we have learned the basics of the Linux command line, we are going to put everything together to download, setup, and administer our own low interaction SSH honeypot. As we discussed in this course, low interaction honeypots provide the smallest amount of interaction with the attackers. In the case of SSH honeypots, a low interaction honeypot may only provide a login prompt to attackers in order to record basic information about each attacker and the credentials they tried. We will start by discussing what SSH is, then we will learn how to install the tool Docker, and finish by setting up our Honeypot.

2.1 SSH

The Secure Shell Protocol or *SSH* is a networking protocol that allows users to connect to a remote computer over a secure channel. When you connect to a SSH server, you will be presented with a command line to execute the same commands we discussed in part 1. Since compromising a host over SSH allows for access to all programs and files of a particular user, attacks against this protocol are very common. By default, SSH servers listen on port 22. Therefore, in order to receive the largest amount of attacker traffic to our honeypot, we are going to want to direct our honeypots to listen on that port as well.

2.2 Docker

Docker is a tool that allows developers to package their programs into portable containers. For the sake of this course, all you need to know is that Docker removes all of the complexity in setting up an application. We will not go into any more detail on Docker (aside from a few basic commands), although I encourage you to continue reading up on Docker as it is an industry standard in the deployment of web applications.

To install Docker, we are going to use the apt tool we discussed earlier. We first want to update apt to make sure it downloads the latest version of each application we request. We do that using the following command:

```
user@wse380-22fa:~$ sudo apt update
```

We then install Docker simply by using the command:

```
user@wse380-22fa:~$ sudo apt install docker.io
```

We have now sucessfully installed Docker!

2.3 Installing a Low Interaction SSH Honeypot

Now that we have Docker installed on our machine, it is easy to spin up a low interaction SSH honeypot. We will be using the honeypot from this Github repository. Since we installed Docker however, we do not need to download this repository directly. Docker is able to automatically pull the code of this honeypot and start it up with one command.

```
$ sudo docker run --name ssh_22-10-06 -d -p 22:22 justinazoff/ssh-auth-logger
```

Here, we are telling docker to pull the application from the repository at "justinazoff/ssh-auth-logger" and run it. We name our container "ssh_22-10-06" to make it easier to run future commands on it. In order to distinguish the data gathered between different executions of our honeypot, we add today's date to the name. We also specify the argument "-d" to tell docker to run this process in the background since we're not interested in interacting with it. Hence, when you run this command you won't see anything happen on the screen. Lastly, we need to tell Docker which port we want to run this process on. For simplicity, we will not go into exactly what this argument is saying, but we are telling Docker to run this honeypot on the SSH port of 22.
Once you have executed this command, you have successfully started your honeypot! It is now listening for connections from attackers. We are going to test it out by connecting to it ourselves. To do so, we are going to use SSH to connect to localhost. In computer networking, localhost is the name of your own computer's private networking interface. In other words, we are connecting to our own computer.

```
$ ssh test@localhost
```

If you see a SSH login prompt after running the above command, congratulations, your honeypot is running! After running our honeypot for a while, we're going to want to see the data it produced. To do so, we are going to use Docker's logging functionality. To see our honeypot's logs, run the following command:

```
$ sudo docker logs ssh_22-10-06
```

Note how we reused the name we gave to our process earlier, "ssh_22-10-06". Running the above command should print a lot of data to the terminal. If you look closely you will see fields such as IP addresses and credentials tried. We are not too worried about the exact contents of this data at the moment but next week we'll extract this data and begin to analyze it. For now though, let's save this data into a file to use later on Do this by running the following command:

```
$ sudo docker logs ssh_22-10-06 > ssh_22-10-06.json 2>&1
```

Note that this is the same command we used to view the logs, but this time we are utilizing the Linux command line's redirection feature that we discussed earlier. This honeypot saves its logging data in a format called JSON. The format itself is not too important, so don't worry if you're unfamiliar with it. We will discuss it next week when we begin to analyze our data.

NOTE: Do not stop your honeypot until we collect more data and begin to analyze it in a future session! If you do want to stop it, make sure you run the command "sudo docker start ssh_22-10-06" afterwards to restart it, and verify that you can "ssh test@localhost" again.

When we decide that we have collected enough data in the future, we can stop our honeypot by running the following command:

```
$ sudo docker stop ssh_22-10-06
```

This will stop the honeypot and clean up all data associated with it.

3 Next Steps

You've now successfully setup your first honeypot on a server in the cloud! Let the honeypot run until our next session where we will look into analyzing the data you've captured.