Systems Security 2017/18 pre-lab

This pre-lab will prepare you for the four systems security labs.

In all the following examples, things you can type on the command line are set in **`bold monospace`** and parameters that you can substitute are in square brackets, so **`cat [filename]`** means that you can type **`cat index.html`** if you have a file index.html.

1. Install the lab VM

The VM that we'll be using is available from the following locations:

- (preferred) during the pre-lab, from the USB sticks provided
- from http://www.cis.syr.edu/~wedu/seed/lab_env.html
- Make sure to download **SEEDUbuntu12.04.zip**

Unzip the SEED virtual machine zip file – you will find a directory containing lots of files that represent parts of a virtual hard disk.

This VM contains a particular version of a 32-bit operating system – your own machine is likely to be 64-bit, and so use different calling conventions (x86-64 has more general-purpose registers, so there's less need to use the stack) and therefore the labs will not work on your machine directly!

To run the VM, we recommend the free virtualbox software. Although it is available in most linux distributions as a package, the ubuntu version has some known bugs – we recommend using the .deb package available from https://www.virtualbox.org/. You should be able to double-click the .deb file to install it.

You cannot run virtual machines on the CS lab computers, so you must use your own computer for the systems security labs.

In virtualbox, follow these instructions.

1. Create a new virtual machine.
2. Set RAM to 512MB – this is enough for our VM.
3. When you are asked to add a hard drive, choose "use existing" and select the file SEEDUbuntu12.04.vmdk from the directory where you unzipped the VM.

2. Linux basics

You may want to revisit the notes at http://www.cs.bris.ac.uk/Teaching/tutorials/linux/.
Or the bash-hackers wiki at http://wiki.bash-hackers.org/doku.php which links to tutorials.

There will be more material on security-related features of UNIX/Linux later in the unit.

3. Ubuntu "root"

Traditionally, UNIX had one user called "root" (technically, UID=0) who could do anything and normal users/groups, whose access to the filesystem can be controlled by permissions.

Ubuntu departs from this principle – we will see why later on in the unit. Instead, users with the correct privileges can "**`sudo command`**" to run a command as root. For extended administration tasks, "**`sudo su`**" opens a root shell.

- Familiarise yourself with the differences between the su and sudo commands.
- How is access to the sudo command controlled (look up "**`man sudo`**")?

4. Package management

Linux software and updates are delivered in packages from repositories, administered by a package manager. We will consider the security implications of this system later on.

Generally, all package management requires root access.

Beginners should install synaptic ("**`sudo apt-get install synaptic`**", then "**`sudo synaptic`**") which is a GUI to the package management system. Ubuntu uses the APT package management system, so the command "**`apt-cache search [keyword]`**" looks for packages whose name or description contains the keyword. Apt-cache does not need sudo as it only reads the package database.

"**`apt-get`**" performs package management operations, and requires root access.

`apt-get update`	fetches the latest package lists from the repositories.
`apt-get upgrade`	installs the latest versions of all packages already installed.
`apt-get install [name]`	installs the named package(s).
`apt-get remove [name]`	you can probably guess this one.

See "man apt-get" for more information. During this unit, you may want to install additional software on the lab VM.

5. Getting programs to talk to each other

UNIX design principles say that each program should

- do one thing well
- work together nicely with other programs
- when it has nothing important to say, keep quiet

Command-line oriented programs tend to read from standard input and write to standard output (and standard error). For example, **`cat [filename]`** reads a file and prints its content to standard output; **`cat`** on its own reads from standard input and prints to standard output. If you call **`cat`** on its own, it does not get stuck – your terminal is waiting to read a line, which will then be output again. Pressing Control+D sends an end-of-file to cat, which then terminates.

grep [expression] [filename] reads from a file and outputs only those lines matching the (regular) expression to standard output; **grep [expression]** reads from standard input.

If a program expects a filename to read from or write to, the files /dev/stdin or /dev/stdout can be used to make it work interactively – or with a pipe etc.

5.1 Redirects

The command **[command] < [filename]** runs a command and feeds it the contents of the file given, so **cat < [filename]** is the same as **cat [filename]** (well, almost[1]) and **cat < /dev/stdin** is the same as **cat**.

[command] > [filename] redirects the output. This will overwrite the file if it already exists; **[command] >> [filename]** appends to the file if it already exists (both variants create the file if it doesn't exist yet).

Sometimes you want to save the output in a file and watch it in your terminal. The command **tee [filename]** reads from standard input and writes to both the file and to standard output – mostly useful within a pipe.

Sometimes you don't care about a command's output – in this case, you can redirect it to /dev/null which is a special file that ignores anything written to it.

5.2 Pipes

A pipe connects the standard output of one program to the standard input of another. So, **cat [filename] | grep [expression]** has the effect of printing only those lines in the file that match the expression, which is the same as **grep [expression] [filename]**.

- What does the following do?

```
cat logfile | grep web | tee log1 | grep error | tee log2
```

5.3 Named pipes

Named pipes or FIFOs (first in, first out) are special files that allow two processes to communicate in one direction – the first process can send data to the second.

If a process writes to a FIFO from which another process is reading, the data gets sent from the writer to the reader. Writing a FIFO that no-one is reading, or reading a FIFO that no-one is writing, blocks until another process performs the opposite operation. Closing a FIFO on the writer side sends end-of-file to the reading side.

Open two terminals to try out the following. The command **mkfifo [filename]** creates a named pipe; **echo [words]** prints words to standard output.

---

[1] In "cat filename" the command (in this case cat) can be aware of the file's name as well as its contents, if it wants to. For example, "grep -nH [expression] *.txt" prints out all matches prefixed by the name and line of the file in which the match was found, for which grep needs to know the name of the file.

terminal 1	terminal 2
$ mkfifo mypipe **$ echo hello > mypipe** (blocked)	
	$ cat mypipe hello
$ (prompt reappears)	**$** (prompt reappears instantly)
	$ cat mypipe (blocks)
$ cat > mypipe	
any line you type into this terminal will now appear in the other. Enter Control-D on a new line to close the pipe.	

5.4 Piping from other programs – popen

The *open* system call opens a file for reading/writing and is implemented in some form in most POSIX-inspired languages. Once you have a named pipe, you can open it like any other file; there is usually some form of the *mkfifo* call to create it.

The *popen* call is usually quicker to use. It runs a process and sets up a pipe from or to it.

```
FILE* popen(const char *command, const char *mode)
```

where *mode* is "r" to read or "w" to write and *command* is any shell command. Close the pipe again with *pclose*. The following example is adapted from *http://pubs.opengroup.org/onlinepubs/009695399/functions/popen.html* and opens a pipe to the "**ls ***" command, reads its output then prints it to stdout.

```
#include <stdio.h>

FILE *fp;
int status;
char path[PATH_MAX];

fp = popen("ls *", "r");
if (fp == NULL) { exit(1) };

while (fgets(path, PATH_MAX, fp) != NULL)
    printf("%s", path);

status = pclose(fp);
```

5.5 Subshells

Sometimes, we just want to execute a program and then grab its output.

```
$ echo "This is $(uname -a)"
This is Linux icy.cs.bris.ac.uk ...
```

The **$(...)** launches a subshell (in which one can use pipes and redirects) and returns its output. An older syntax for subshells uses the backtick (`) character, i.e. **echo "This is `uname -a`"** does the same as our example, but the newer syntax can be nested[2].

6. Getting programs to talk, advanced version

In addition to pipes there are two more ways to get programs to talk to each other – or to script one program from another – sockets and pseudo-terminals.

6.1 Sockets

You will have encountered TCP sockets in network programming (a host:port pair such as www.google.com:80 is really a socket) but UNIX also allows you to create domain sockets, that have filenames instead of port numbers. The advantage of sockets is that they allow bidirectional communication.

Here is a simple echo server in python that listens on TCP port 8000.

```
import socket

mysocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysocket.bind(("", 8000))
mysocket.listen(1)
while 1:
    conn, addr = mysocket.accept()
    while 1:
        data = conn.recv(1024)
        if data:
            if data.find("END") > -1:
                break
            conn.send("Echo: " + data)
    conn.close()
mysocket.close()
```

You can run this server and use telnet localhost 8000 to connect to it from a different terminal:

```
$ telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
foo
Echo: foo
```

[2] Backticks can be nested too by escaping the inner ones, but this produces hard-to-read code.

```
bar
Echo: bar
THE END
Connection closed by foreign host.
$
```

Since the server does not have a shutdown command, use Control-C in the server's terminal to kill it. And here is a TCP client:

```
import socket
import sys

mysocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysocket.connect(("localhost", 8000))
mysocket.send("hello")
response = mysocket.recv(1024)
print "Server said: " + response
mysocket.send("END")
mysocket.close()
```

Running this while the server is active should print "Server said: Echo: Hello" on the client terminal. The client and server can communicate back and forth multiple times.

And now, the same with UNIX domain sockets. Change the two lines in the server to read

```
mysocket = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
mysocket.bind("./socket")
```

and do the same for the client. Your client and server can talk to each other via the UNIX socket created in the special file *./socket* . Check the directory with **`ls -l`** and you will see that the socket has type "s".

Note: after running the server, you must manually remove (rm) the socket. You can only bind to a TCP socket that is not in use, or a UNIX socket file that does not exist yet.

From a security perspective, a UNIX socket is a file so it can have access rights like any other file – and only users with access to the filesystem in question can use it, unlike a TCP socket that (without firewalls) is wide open to the world.

6.2 Pseudo-terminals

Sockets are nice but we have to invent our own communications protocol: in the example above, we used the word END to indicate that a conversation is over and expected every message to fit in a single 1024-byte buffer.

- Recall how HTTP deals with this problem?

Pseudo-terminals or PTYs are more powerful abstractions. The terminal you have been using so far (xterm, gnome-terminal or similar) implements a pseudo-terminal internally to communicate with the bash shell that you have been using. The secure shell (ssh) is another use case for a pseudo-terminal to connect to a shell on a different machine.

We can use PTYs ourselves to script a program that reads from standard input and writes to standard output. Since they can be quite fiddly to use, we will use python's pexpect module that wraps all the nasty bits for us.

To run a program and get its output, we use the run method – the equivalent of a subshell :

```
import pexpect
result = pexpect.run("echo hello")
```

The variable result now contains something like the string 'hello\r\n'.

The calculator program bc reads a line such as 1+1 from standard input and writes the calculated result, in this case 2, to standard output. Let's script this.

```
import pexpect
p = pexpect.spawn("bc -q")
p.sendline("1+1")
p.readline()
result = p.readline()
print "bc says: " + result
p.close()
```

The first *readline()* gets the echoed line 1+1, then we read the next line to get the result 2. We could also have written *p.setecho(False)* to turn off the echo.

The name *pexpect* comes from the fact that one can script interactive programs by waiting until an expected string occurs. For example,

```
...
p.expect("Username:")
p.sendline("david")
p.expect("Password:")
p.sendline("123456")
p.expect("Login successful.")
...
```

7. Let's break something!

Grab the files access.c, md5.c and md5.h from the unit website (labs page). Compile with

```
gcc -oaccess access.c md5.c
```

It asks for a username and 4-digit PIN and prints either “Success” or “Incorrect”.

From looking over a colleague's shoulder, you know that their username is *david* (not capitalised) and their password is a 4-digit number ending with 8 (you only saw the last number of their PIN). You could try 1000 combinations by hand ... or you could write a script to do it for you.

- Find the PIN – for example by calling the program in a loop, trying all remaining 1000 possibilities. Use any language and tools that you like.

8. (optional advanced topic) – socat

socat is a tool to get different programs to talk to each other if they were not written to use the same kind of channel. For example, it can connect a PTY to a socket. The manual page at http://www.dest-unreach.org/socat/doc/socat.html explains the many options.

Note – the debian/ubuntu version of socat does not support readline, because of a license incompatibility with OpenSSL. Download the original source and recompile to get a fully functional version of this useful tool.

Some examples:

```
socat READLINE TCP4:www.example.com:80,crnl
```

Like telnet, but you get readline features (up/down keys to select previous lines etc.) Translates the shell's newlines correctly into the CR,NL that the web expects.

```
socat READLINE EXEC:program
```

Readline wrapper around arbitrary programs.

```
socat TCP4-LISTEN:9000,fork EXEC:sh
```

Slightly evil. Anyone connecting to port 9000 gets a shell (with the current user's privileges). Don't try this on an internet-connected machine.

```
socat TCP4-LISTEN:9000 TCP4:myserver.example.com:80
```

Port forwarding.

```
socat - \
SSL:myserver.example.com:443,cafile=myserver.crt,cert=client.pem
```

Connect to a SSL/TLS secured server, in this case opening an interactive session (-), use certificate pinning (only accept certificates in the .crt file) and provide a client certificate (pem file) if required. Great for testing (socat can also be a SSL server using SSL-LISTEN).