CSci 4271W (011 and 012 sections) Lab Instructions

Lab 4 **February 17th, 2025**

Ground Rules. You may choose to complete this lab in a group of up to three students. Before you leave the lab, **make sure you have submitted to Gradescope, you included all group members on the submission, and the autograder found all required files!**

1 Lab 4: Just Passing Through. . .

In this lab, we'll look at a very common vulnerability in programs that run other programs. Login to your CSELabs workstation, or if you're completing the lab remotely, login to VOLE, then open a terminal by right-clicking in the desktop and selecting "Open Terminal Here." In the terminal, connect by ssh to your VM as we did for labs 1 and 2. In vole or on your desktop, use your favorite text editor to start a file keeping your notes, called `lab4_notes.md`.

1.1 Connecting commands together

In lab 2, we saw that the pipe operator – | – can be used to connect the output of one unix program to the input of another. It turns out that the standard unix command shells include several other methods to combine the execution of two programs:

- The simplest is the semicolon `;`. Running the command `cmd1 ; cmd2` runs `cmd1` followed by `cmd2`, for example try running the commands:

```
$ cat /etc/timezone ; echo "done!"
$ echo "sleeping now..." ; sleep 10 ; echo "okay, awake again!"
```

- Two methods that are closely related are || and `&&`. In Unix-style OSes, every program exits with a "status" which is either 0, meaning basically that the execution was "successful" or not zero, meaning that there was an error. You can check this status immediately after by accessing the shell "variable" `$?`, for example, try the following sequence:

```
$ cat /etc/timezone
$ echo "etc timezone result: $?"
$ cat ajksdfjk
$ echo "non-existent file result: $?"
```

The command line `cmd1` || `cmd2` determines if `cmd1` OR `cmd2` would be successful:

```
$ echo "always succeeds" || echo "always"
$ echo $?
$ cat dlksdgkj || echo "always succeeds"
$ echo $?
```

Notice that in the first case, the second command doesn't run, because much like your favorite programming language, the shell's boolean OR operation "short circuits" if the first command succeeds. Similarly, `cmd1 && cmd2` determines if `cmd1` AND `cmd2` would be successful. Before you run the following commands, see if you can predict what you will see as the output:

```
$ echo "always succeeds" && echo "always"
$ echo $?
$ echo "always succeeds" && cat aklsdfajkl
$ echo $?
$ cat alskfakj && echo "always succeeds"
$ echo $?
```

In your lab4_notes.md file, record your predictions, the actual result, and your explanation for why.

- Another method to combine commands is argument substition via the “backtick” operator:

```
$ cmd1 `cmd2`
```

Runs cmd2, and sets its output as an *argument* to cmd1. An equivalent syntax for this operation that is sometimes easier to read and type uses a dollar sign and parentheses:

```
$ cmd1 $(cmd2)
```

So for instance, try to predict what each of the following command lines will result in:

```
$ echo `cat /etc/timezone`
$ echo "Today is" $(date +%A)
```

Record your predictions and the results for each of these command lines as well.

1.2 Running programs within programs

Depending on the programming task, it is sometimes useful for one program to run another program and either write to its input, read from its output, or access its exit status. The C standard library provides many ways to do this but the two most common are the `system()` and `popen()` functions. (In Java, the `Runtime.exec()` method wraps calls to `popen()` and in Python3, the `subprocess.run` and `subprocess.Popen` commands do the same thing; in PHP, they are `system`, `popen` and `proc_open`)

If a C program calls `system("command to run");` the operating system will open a separate shell and run `command to run`, and the exit status will be the return value in the C program; `popen("command to run", "rw");` also opens a separate shell running `command to run`, but it returns a `FILE *` pointer that the C program can read to or write from to get input/output from the command.

1.3 Why am I learning this in a security lab?

Well, a very common source of elevation of privilege errors is when a program passes input across a trust boundary into a `system` or `popen` command. For example, if an email contains a link to a webpage and the mail client opens links by using `system` to run a browser, then a link like `<a href="http://example.com; rm -rf ~">` could result in the mail client calling `system("/path/to/browser http://example.com; rm -rf ~")` and the shell would run the browser, then run `rm -rf ~`. This is often called a "pass-through" attack, as the adversarial command (`rm -rf ~`) is "passed through" to the command line. (A pass-through attack involving a Java logging library was the underlying case of the "log4shell" security incident that caused massive outages in services such as Cloudflare, Steam and Minecraft in December 2021.)

For this lab, we'll look at two programs that use these functions in a way that is vulnerable to pass-through attacks.

1.3.1 wave

The first program, `wave`, is a program that will send a "wave" message to all of the terminals a given user has open on a particular computer. To work correctly, it would need to be installed "setuid root" so that it would run as the "root user" (this is a concept we'll talk about more when we discuss OS security mechanisms), so getting the program to run an unintentional command would definitely represent an elevation of privilege attack. For this lab, though, we won't go through the extra steps to do that, we'll just verify that we can use inputs to make it run code.

To get started, clone the lab4 git repo using

```
$ git clone https://github.umn.edu/badlycoded/passthru.git
```

and then cd into the `passthru` directory to see the source code `wave.c` (This is a buggy program, don't install it on a non-virtual computer.) Go ahead and look through the code (for example, using `less`)- you'll see that it uses a pipe to run the `ps` command (which lists what programs are running on a Unix computer) and searches for the username listed in the file provided as a command-line argument. If you compile the program and then run it with the command-line argument `student.txt` (so waving to yourself...) you should see:

```
$ gcc -Wall -g -o wave wave.c
$ ./wave student.txt
Waving...
Wave! from student
Sent!
student@csel-xsme-s25-csci4271-NN:passthru$
```

Suppose we wanted to make the `wave` program (running as root) write to a file we can't access (like the password file, for example, or the `.ssh/authorized_keys` file of a different user...). To be concrete, imagine that what we want to do is have the program write `Pwned!` to the file `pwned.txt`. If we were to do this at the command line, we would just run the command `echo "Pwned!" >pwned.txt` but the objective here is to get `wave` to do it for us. See if you can work out how to give an input file, `pwn_wave.txt`, to `wave` that will accomplish this task. That is, you will run `./wave pwn_wave.txt` at the command line, and afterwards, a new file named `pwned.txt` should exist that did not before. Here are a few extra things you might find it useful to know:

- You can include spaces in a command line argument by using single or double quotes, e.g. `'this is a single argument'` and `"so is this"`.
- command lines can have comments: like in python, the `#` character turns the rest of a command line into a comment:

```
$ echo "I like having 'quotes' in my arguments" # but not the rest of this
```

Keep a record of the contents you tried for `pwn_wave.txt` in `lab4_notes.md`, why you tried them, and what the result of each attempt was. (Make sure you check for the file `pwned.txt` after each attempt! Getting an error message or seg fault doesn't necessarily mean that your *attack* failed!)

Note: if running `wave` with your "exploit" input still results in `wave` printing `Sent!`, or never prints `Waving...` then you are probably executing the command in "your" shell rather than the one invoked by the wave executable, and thus not succesfully executing the intended attack. Your TAs can help you determine if this is the case, if you're unsure.

1.3.2 sendgrades

The second program, `sendgrades` expects to read a `csv` file consisting of student email addresses, grades, and comments, and sends each student an email including their grade and comment. If the person running `sendgrades` did not write the input file, then that is potentially a flow across a trust boundary! We wouldn't want inputs from that flow to reach a command line without being careful to prevent them from being interpreted as commands rather than inputs.

You will also find the source code for sendgrades in the same repository. If you open this file in e.g. `less`, you will see that the program uses the Unix `mail` command-line utility to send these emails, through a call to `system()`. The `sudo apt install` command below will install the `mail` command if you don't already have it.

If you compile `sendgrades` you can make a csv file using `echo` and test it out:

```
$ sudo apt install mailutils
$ gcc -Wall -g -o sendgrades sendgrades.c
$ echo 'abc@example.com,100,good job' > file1.csv
$ ./sendgrades file1.csv
```

Your task for this portion of the lab is to make a one-line file, "evil.csv", that will cause `sendgrades` to run a command of the attacker's choice. For concreteness, let's have our goal be to run the command `ping -c 3 128.101.101.101`, which will show that we can make the `sendgrades` send data to the network (go ahead and run this command line in your VM if you want to see what to expect as output when you succeed). As in the previous example, keep a record of your attempts in `lab4_notes.md`. One thing to keep in mind is that `sendgrades` does use single quotes (`'`) to preserve spaces in the values it passes to `system` – you may have to "close" this quote in your input file. Another thing to keep in mind is that `mail` might fail due to an invalid input format; you might have to be a little "clever" about where you inject the command in evil.csv. Explain in your lab notes how your final `evil.csv` file results in the call to `system()` running `ping`.

1.4 All done!

Once you've got your `pwn_wave.txt` and `evil.csv` files working, use `scp` to copy them to the same machine where you've been keeping your `lab4_notes.md` file. Now you are ready to upload the files to gradescope. Go to the Lab 4 assignment in gradescope, and click in the "DRAG & DROP" box to (multi)-select the `pwn_wave.txt`, `evil.csv`, and `lab4_notes.md` files from your desktop and upload them. Make sure you include all of your group members in the submission!

Once you've submitted the file, the autograder will test the to make sure the proper files were submitted, and notify you if either are missing, within a few minutes. We'll manually review your notes file and the `pwn_wave.txt` and `evil.csv` files, and as long as you managed to "exploit" one of `wave` and `sendgrades`, you'll receive full credit for the lab.

Congratulations, you've finished Lab 4!