CS 481 Lab 1

Due by 11:59pm on Monday, 20 February, as an e-mail to the instructor (jedcrandall@gmail.com). Please send only PDF files.

100 points (out of 100 total for Lab 1)

The purpose of Lab 1 is to understand what processes are in a Linux system, including their interactions with the system (specifically their virtual address space, file descriptors, interprocess communication, and system calls, and Linux's security mechanisms).

In your writeup, you should tell me everything you can about an Ubuntu 10.04 Server 32-bit system. There is no page limit, I'm guessing that with all the figures and everything some writeups will be more than 50 pages, but it's also maybe possible to answer all these questions and tell me everything else I need to know in 20 pages or less. I'm not so concerned about the length of your writeup, I'll be looking for completeness.

Figures and graphs and such are much appreciated, but are not required. The main thing is to get the basic structure of what's going on across to me effectively.

You may use any programming language you like, I recommend a scripting language such as Perl or Python.

You may discuss general information with your classmates and with others, but the results you report in your writeup should be your own that you produced from your virtual machine. Discussions should be limited to general things, such as posting to the cs481-chat mailing list a question like, “Has anybody figured out how to know what signal handlers a process has registered?”

I expect you to consult Bovet and Cesati a lot during lab 1, and explain what the kernel is doing in all parts of your writeup. This is an operating systems class, not a UNIX administration class, so make sure to always keep your mind on the kernel even thought most of the information you'll gather for this lab will be gathered in user space. For this lab, you should combine the information you gather with your readings in Bovet and Cesati to explain what is going on in the kernel and at the boundary of the kernel and user space.

Be sure to explain your methods in all parts of your writeup. How did you get the information? How did you aggregate it if you're reporting an aggregate value? There should be enough detail that someone could retrace your steps and get the same result. You should get most of the information through your own virtual machine, if there's any information that you get from another source that you could have gotten through your own virtual machine that won't count (and remember that this may be

considered cheating if you attempt to represent that information as your own work). Be sure to ask me if you have any questions about cheating or collaboration.

If you do not want me to share your lab 1 writeup with others (such as showing it to the class as a good example or giving it to other students in the future who are curious about Linux), please indicate this clearly at the top of your writeup.

Part 1: Overview

Start with a brief description of the system. How many processes are running? Who owns the processes? How many file descriptors are open total? How much memory is the system using? And so on...

Then give me some idea about the tree of processes, including the permissions and the role of each process. I don't need the full tree, just the root of the tree and the immediate children of the root and enough branches to give me some idea of how a shell gets created when someone logs in and then when they run a program from the shell how that looks in the tree, *etc.* What file descriptors seem to be inherited from parents in the tree? You should compare two or three different ways of logging into the system and getting a shell, such as the terminal, the network over SSH, or a serial connection.

Now, try to explain the memory usage of the system. How much memory is being used for shared libraries? How much total memory are processes using that's not shared? Can you reproduce the same number that top shows in terms of memory usage by accounting for all the shared memory?

Next enumerate all the different kinds of Interprocess Communication that you can find, giving examples of running processes for each.

Then tell me about several interesting processes. Are there any processes that register interesting signal handlers? Are there any multi-threaded processes? Are there any processes with interesting or anomalous virtual address spaces (*e.g.*, a process that doesn't have a heap)? Are there any file descriptors that stick out as anomalous or interesting? What processes have accrued the most runtime? The least? What processes do the most context switches?

Part 2: An Unnamed Pipe

In part 2, we'll do a system call trace to find out what happens when someone who is logged into a shell types:

```
ls -la /usr/bin/ | awk '{print $1 "\t" $6 "\t" $7 "\t" $3 "\t" $8}' | sort
```

You'll want to `strace` from a different shell by attaching to the shell process and tracing its children, as we did in class.

I want to know everything that happens from the moment the user hits enter until the shell prompt is returned at the end of the command after all the output has been printed. What processes get created? What is their parent? How are the pipes set up? Make sure not to just say which system call is used, but also tell me what the steps the kernel performs to carry out that system call in the Linux kernel source code are.

A caveat to keep in mind: Traditional UNIX systems programming would use a `fork()` system call, whereas what you'll probably see in a modern Linux system is `clone()`. Just keep this in mind and be sure to note it in your writeup. If you're ever having a discussion with a UNIX old timer, they might not know what you're talking about if you say `clone()` instead of `fork()`. I don't know if the shell explicitly uses clone or if the glibc `fork()` library call is just a wrapper for the `clone()` system call or what, if you'd like to solve this mystery for me using `ltrace` that will be worth a couple of bonus points.

Some specific questions you should be sure to answer for this part (but this is not an exhaustive list, make sure to tell me everything I need to know even if there's not a question here for it):

- What do the child processes' address spaces look like just before the `execv()` system call? What do they look like after? How does the child's address space (heap, stack, libraries, *etc.*) get set up? What data structures does the kernel keep to keep track of these objects in the process' address space?
- Most binaries such as `sort` and `awk` look for their input on stdin by default, how are special system calls, inheritance of file descriptors, and closures of file descriptors used by the shell to create the pipes that the user wants so that the output of one child becomes the input of another?
- How many system calls are made before the main program of a child binary is actually executed? What are all these system calls doing? At what point does a child process cease to be running code from the shell binary and start running code to become a new binary?
- How does a command name like `sort` actually get associated with a binary on the filesystem that is mounted and executed? What checking and searching does the kernel do? What parts of this process are done in user space?
- How do the command line arguments that the user typed in the shell end up being passed as function arguments to `main()` in the child process? Hint: there are several steps here, and the kernel is involved.
- Is there anything keeping the shell from writing output to the screen while the command is still running? How does the shell know when the child processes have all finished?

Part 3: Interrupts and Exceptions

In this part, we want to solve a couple of mysteries about things that you've probably seen repeatedly since you started programming: hitting Ctrl+C to get out of an infinite loop and SEGFAULTs.

First, write a program that causes an infinite loop. Do a system call trace on this program, or use some other mechanism for knowing what signals are sent and which ones get handled, and by whom. The keyboard interrupt when you hit Ctrl+C should send an obvious signal to the child, but what other signals get sent and why?

Next, write a program that causes a SEGFAULT. Using system call traces and/or other tools, explain everything that happens between when the process tries to access memory it shouldn't and when "Segmentation fault" is printed on the screen and the shell prompt is returned. Also be sure to explain the process the kernel goes through, and why it makes the decisions it makes throughout the whole sequence of events. What process is printing "Segmentation fault" on the screen?

Part 4: Security

From a regular user shell, you can get a root shell by typing:

```
sudo su -
```

Use system call traces to explain how it's possible to get a root shell from a shell process that does not have root privileges itself. Hint: it has to do with the set UID bit of the `sudo` binary.

Also explain why, unless there is some security vulnerability in the form of a software bug or misconfiguration of the system, someone needs to know a password for a user in the admin group to get a root shell. (Note that many UNIX systems have a root user password and a wheel group, but in Ubuntu the default is that the root user has no password and therefore no way to log in directly).

Briefly explain the foundations of security in a typical UNIX system. This should be about one to two pages, I would estimate. Explain the roles of directory structure, the process hierarchy, special file attributes such as the SUID bit, and other things you can find out to show how one can reason about UNIX system security. What keeps someone from compiling code to become root who doesn't have administrator access? What kinds of tricks (probably involving software bugs) could someone without a password for administrator-level access use to get a root shell?

Be sure to start early and ask lots of questions. If you don't understand the assignment, that's okay, I expect that most of the class will need to ask a lot of questions before understanding what needs to be

done for this assignment, and then ask many more questions along the way. Here are some hints to get you started...

```
less /proc/123/status
```

```
less /proc/123/maps
```

Try reading the man pages for `fuser`, `lsof`, `strace`, `ls`, and `man`, particularly you should note that the man pages have different levels, *e.g.*:

```
man 7 signals
```

```
man 5 passwd
```

```
man 2 open
```

```
man 7 pipe
```