CSE 509 Assignment 2: System-call and Library Interception

1 Background

System-call interception provides a versatile mechanism for a number of security-related applications. The most obvious one is policy enforcement: one can define policies that govern which system calls are permissible for a process and which ones aren't. Alternatively, one can use system-call interception for transparently extending application functionality. For instance, you can extend an arbitrary application so that it can access remote files using the HTTP protocol. This can be done by intercepting open system calls, identifying file names that correspond to remote URLs, making an HTTP request to the remote site to fetch the content. Subsequent reads on the same file descriptor should be intercepted, and this remote data returned to the process.

There are several approach for doing system-call interception. One of the mechanisms available for this purpose on Linux is the `ptrace` mechanism. You can find out more by doing `man ptrace` on a Linux system. There are also lecture notes on system call interception that describe this system-call interception mechanism. To learn more about the ptrace interface, look at the following references:

- `http://www.linuxjournal.com/article/6100` is a link to a good article in the Linux journal that explains ptrace.
- `man ptrace` on Linux systems
- `strace` is a handy command-line tool that uses ptrace to intercept system calls, and prints them. You can understand how ptrace works by running `strace` on various programs. You are also welcome to read its source code, but don't copy it.

While ptrace provides a secure mechanism, it introduces significant performance overheads. An alternative approach that has low overheads is the library interception approach. This approach is based on the fact that system calls are usually low-level mechanisms that are not directly used by applications. Instead, applications call system-call wrappers in standard libraries (such as `glibc` on Linux). If all calls to these wrappers can be intercepted, then we will have a much more efficient approach. However, there are serious drawbacks as well — in particular, a malicious program can bypass these wrappers and make a direct system call (using a software trap instruction). So, this approach can be used only with benign applications. To use this approach, you can develop a library that provides a routine for each system call that you want to intercept, and using `LD_LIBRARY_PATH` mechanisms to ensure that your library comes before the standard system libraries. Here are a couple of references on the topic:

- `https://blog.netspi.com/function-hooking-part-i-hooking-shared-library-function-calls-in-linux/` illustrates library hooking using `LD_LIBRARY_PATH` technique, and illustrates it with examples.
- `http://samanbarghi.com/blog/2014/09/05/how-to-wrap-a-system-call-libc-function-in-linux/` describes the `LD_LIBRARY_PATH` technique, as well as a second, finer-granularity approach for wrapping individual library calls.
- `ltrace` is a command-line tool very similar to `strace` but can report calls to dynamic libraries instead of system calls.

Note that if you want to use library-based interception with untrusted code, then you need to use additional mechanisms to guard against bypass.

2 Assignment Description

2.1 Warmup (40 points)

This part of the assignment is NOT meant to be a group effort. You should work independently. The remaining parts of this assignment can be completed by groups of two, similar to the first assignment.

1. (15 points) Get familiar with `strace` by reading its man page and by using it. What are the system calls made by `ls` when you run it with no options? What about `ls -l`? Try the command on directories large and small to have confidence that you have exercised `ls` well. Can you explain the reasons for differences in system calls made?
2. (15 points) Get familiar with `ltrace`. Use it to study `ls` in the same was as the problem above. Use appropriate options to `ltrace` (or process the output using tools such as `grep` or `awk`) so as to leave out calls to utility functions such as malloc, free, getenv, getopt, strcmp, and so on, while only retaining calls that look like system calls. (But **do not** use the `-S` option to `ltrace`.) Comment on the output you get this way, as compared to `strace` output.
3. (20 points) Use `strace` to count the number of files accessed by applications when they are started up. Answer this question for `ls`, `nano`, Open office and Firefox or Chromium. Note that complex applications may create multiple child processes. Use appropriate options to `strace` so that it captures the system calls made by all of these processes. Note that applications such as Firefox open thousands of files on start up; if you don't see this, then you are making some mistake.

Your submission for this part will be a PDF document that reports on your findings. Include all the relevant details, but don't go overboard. Ideally, each report will be half a page or less of text. (If you include the output of some of your tests, make sure that does not go beyond a couple of pages at most.)

2.2 Transparent Application Functionality Extension (60 points)

An important use of system call (or library call) interception is to extend the functionality of applications *without requiring modifications to applications.* This extension may or may not be security-related. An example of a security extension is one that limits access to files, e.g., prevents files in your home directory from being overwritten. You will explore such extensions in this assignment.

*It is silly to extend **toy** applications this way.* So, don't test your implementation on toy applications. For grading, we will test against arbitrary applications, including complex applications such as gedit. Some of these applications will create child processes, and *your extension should automatically trace all of those descendant processes.*

Choose **one** of the following extensions for your assignment. Make sure that you read and understand each part in detail so that you can make an informed decision that maximizes the points you get on this assignment.

- Write a ptrace-based extension that enables applications to access remote URLs as if they were local files. For instance, using your `url2file` tool, I should be able to run a command such as

  ```
  url2file wc https://www.cs.stonybrook.edu
  ```

  to count the number of words on a web page. Before you start writing code, use `strace` to identify which system calls you need to handle. For instance, many applications use `stat` before attempting to `open` a file, so you need to intercept `stat` in addition to `open`. When an application does open an URL, your extension should use a program such as `wget` to download the web page into a temporary local file. From this point, you should arrange it so that `read` operations on the "file" go to this temporary file.

  For full credit, get `url2file` to work even when the local application attempts to load or execute a file. Make sure that you test file loading in addition to exection. (Transparent downloading and execution of remote code is out right dangerous, so don't do this outside of the VM you are using for this assignment.)
- Write an extension `safex` that sandboxes applications so that they can be executed safely. For the purposes of this assignment, you can interpret safety to apply exclusively to file accesses; you need not concern

yourself with the safety of other types of operations, e.g., network accesses. Specifically, you can interpret safety to mean (a) the application can open any file for reading if the file name does not appear in the file `~/.safenoread`, and (b) if it opens a file for writing, a temporary copy of the file should be made, and the application's access redirected to this copy. You can use the library function `mkstemp` to create this temporary name without making unsafe assumptions that can lead to file-name based attacks. The purpose of (b) is to ensure that the application won't fail because of a denial of write access.

Your submission for this part will be a tgz file that we should be able to untar on Linux. We should be able to run make inside the extracted directory to build your program, and then execute that program.

If you run out of time and are not able to complete the assignment, make sure that you have a submission that still compiles and works partially. Typically, such a partial implementation will not be able to handle all the relevant system calls, but is still useful to extend some of the simpler applications. But if your program does not compile or run, then you leave us no option except to give you zero credit. It is your responsibility to ensure that your program works on the VM that was used for your first assignment.

2.3 Extra Credit: eBPF Filters (35 points)

As discussed in the class, eBPF is a recently introduced capability for adding extensions safely to the Linux kernel. These capabilities can be used to solve problems relating to security, performance, and reliability of applications and/or the whole system, and do so *without* having to make changes to the applications or the OS. Being relatively new, these applications of eBPF have not been explored much, so there are numerous opportunities for research and innovation.

The key feature of eBPF that makes these capabilities within our reach is the fact that the extensions are guaranteed to be safe. This not only means that it is easier to write and debug extensions, but also that users will be much more willing to deploy tools based on eBPF, as compared to other tools that may require changes to the OS code.

This extra credit assignment requires a good deal of reading up and learning new technology. It is meant for those who are intrigued by the possibilities presented by eBPF, but if you are looking for some easy extra-credit points, this will likely end up being far too much effort. If you decide that you are really interested in exploring this, start with one of the following overviews:

- `http://seclab.cs.sunysb.edu/seclab/pubs/rohitth.pdf`: Section 2 of this Master's thesis is an introduction to eBPF.
- `https://ebpf.io/` gives you a quick high-level overview of eBPF.
- `https://filipnikolovski.com/posts/ebpf/` is another useful document that lays out some of the possible ways to use eBPF.
- `https://www.collabora.com/news-and-blog/blog/2019/04/05/an-ebpf-overview-part-1-introduction/` is a more in-depth introduction to eBPF.

The first two are the easiest to read — you should start by reading Section 2 of the first document, and then take a quick look at the second link. If you are intrigued and want to learn more, then take a look at the other two as well.

There are two main ways to write eBPF extensions: one uses a limited version of C, and the other uses a shell-script-like language. Needless to say, the C-interface is more powerful and flexible, but you may find the shell interface to be adequate, depending on what you want to achieve. Both of these are described in the first document above, and you can look up more information easily.

Using one of these two interfaces, you should implement a system for logging the network and file accesses made across applications running on the system. Specifically, you will intercept the following system calls:

- *open, truncate, create*
- *connect*
- *accept*

- *execve*

Look through the list of Linux systems calls to ensure that ***all variants*** *of the above system calls are logged as well, e.g., openat.*

The end output should be a sorted list of accesses made on the system, organized into the following groups: (a) file reads, (b) file writes, (c) file executions, (d) connects, and (e) accepts. For file operations, the absolute path name of the file should be included. (If you are able to capture just the relative path names, you may lose a couple of points.) For connects, the IP address (or the domain name) and the port number of the remote machine should be included. For accepts, include the remote IP address and the *local* port number. Within each group, the names should be sorted in decreasing order of the number of accesses. Your output should have a header line for each group, followed by a list of accesses in that group. Each access should provide the relevant resource information (file name or IP address/port), together with the number of accesses.

eBPF includes map data structures that you can use to collect and organize the information you want to display. If you can figure out how to do this, then your implementation will be very fast. Alternatively, you can record access information to a file, and then post-process the file to generate the desired output.

As for the previous part, submit a tgz file. Make sure that there is a README file in the tarred directory that says how to run your program. This should be just a few lines. Make sure that you indicate (a) how we can run arbitrary applications in the background while your system is still running, and (b) how to stop your system so that it can print the output and then quit.