CS 258, Midterm Exam, Winter 2015

Terms and Conditions. This is a take-home, open-book, open-manual, open-shell exam. The solutions are due by class time on Tuesday February 17. Extensions may be offered due to special circumstances—but expect to get an extra problem or two.

You are expected to use (at least) DTrace, Truss, the OpenGrok source code browser[1] and the Modular Debugger's running kernel inspection capability (`mdb -k`), and any other tools you find useful. The documents in the class directory http://www.cs.dartmouth.edu/~sergey/cs258/ and the textbook's index might be useful, too.

For each problem, you should "show your work": the output of the above tools on your actual platform (virtual or physical).[2] For OS kernel code lines, provide the filename and the line number, or the OpenGrok URL pointing to the right line.

You are allowed to discuss the use of tools with your fellow students, *but not the solutions themselves*. For example, sharing a tracing trick is OK, but sharing part of a solution as such is not. Note that in most exercises you are free to choose your targets (which may help you avoid conflicts with the above rule). If in doubt, ask.

Submit your work as a text or PDF file, or as a tarball (.tar.gz file) that contains a directory named `<YourName>-midterm` (with your solutions inside :)).

Note: The default system for these problems is Illumos, due to the great flexibility of DTrace and MDB. You may choose to do some or all or the problems on GNU/Linux instead of Illumos if you so desire (e.g., using *SysTrace* and *Kprobes*); note, however, that Illumos' tools for examining a running kernel are much more versatile and stable. Linux will likely be more work, unless you've been working on a Linux project idea and practiced with Linux tools already.

All files mentioned below are in http://www.cs.dartmouth.edu/~sergey/cs258/midterm/.

Problem 1. *Sharing is caring.*

This problem assumes a "typical" system load. Start a few applications on your VM's desktop to imitate it.

- Find the process that shares the largest number of its memory pages with others.
- Find the ten most popular shared files on the system (i.e., files `mmap`-ed into the memory spaces of the largest number of processors).
- *Extra credit:* Review the above answers w.r.t. pages not just `mmap`-ed but also resident in RAM.

Problem 2. *You cannot stop me!*

It's customary to interrupt processes by sending them a SIGINT by typing a `Ctrl+C` on the terminal where the process runs. Alternatively, you can send a SIGINT with the `kill -s 2 <PID>` command from another terminal.

Of course, a process might hate being interrupted; so it can block SIGINT or define its own function to be called when receiving it. There are several standard library functions for this, such as `sigignore` and `sigaction`.

[1] http://src.illumos.org/source/

[2] You can record your entire shell session with the `script <filename>` command, or, when working in MDB, with the `::log <filename>` command—but the latter will miss your outputs piped to shell via `!` pipes.

Trouble is, the following fine program forgot to call one of those, and it's already running. Help it (or, rather, its process) ignore SIGINT and keep running despite `^C` and `kill -s 2` (including by root). You are allowed "destructive" manipulations of the kernel state; recall that you can write kernel memory from `mdb -k` with several formats (cf. `::formats ! grep write`) after giving the `$W` command.

```
#include <stdio.h>
#include <unistd.h>

int main(){
  int i = 0;
  while(1){
    printf("%d\n", i++);
    sleep(1);
  }
}
```

Problem 3. *U Can't ~~Touch~~ Attach to This!*

Investigate and describe the mechanism by which the GDB debugger attaches to a running process (`gdb -p <PID>`). In particular, capture and demonstrate:

1. How GDB gets the target process to stop;
2. How GDB gets to read or write the other process' memory;
3. How (and when) GDB sets breakpoints.

What mechanism prevents users other than the process owner and root from attaching to a running process? For example, why can't users from the same group as the owning user attach? Show relevant permissions and where they are created.

Extra credit: Acting "destructively", can you make it so that only root (not the owning user) can attach to running processes? Can you make it possible for the members of the owner group (or everyone) to attach?

Problem 4. *Old McDonald has a dynamically linked farm.*

Download a silly program farm.zip from the class directory. This program demonstrates how a shared object file can be made (see Makefile) and loaded into a running process with `dlopen` and `dlsym` calls. Change `-m32` to `-m64` at the top of the Makefile to switch between compiling the 32-bit and 64-bit versions; do `make clean` in-between!

1. Disassemble the text of the 32-bits and 64-bit versions of the program. Explain how each version of the animal noises functions locates its constant strings. In particular, in the disassembly of the 32-bit version, each function seems to contain not one but two `call` instructions – why?
2. You may notice that the old MacDonald's farm is somewhat messed up: the respective animals make the wrong sounds. Edit the **binary** `animals.so` to set the farm right; do not touch the source or recompile. Your solution should not segfault.

Note: There are many ways to edit a binary file. You can install a dedicated editor such as `hexedit` or `ghex`. I find it most convenient to first convert a binary file into an ASCII text hex dump, edit or search that file in my favorite text editor, and then convert it back to binary, as follows: `xxd animals.so > animals.hex` then `xxd -r animals.hex > animals.so`. Submit your hex dump version or, better, the `diff` of your version and the original.

Problem 5. *A farewell to objects.*

Trace from which `kmem` caches objects are released when a process dies. List the types and counts of objects released. Briefly explain the function of each kind of object (i.e., what was it doing for the process?)

Problem 6. *Hash tables have nothing to lose but their chains!*

1. Estimate the average length of collision chains in `/proc`'s hash-by-pid table (`pidhash[]`) under a typical system load.

2. Same for the `page_hash[]` table.

Extra credit: Create a visualization of what the `procdir[]` process table look like at the point when you run a `ps` command.

Problem 7. *Output wants to be free!*

A process is writing to `/dev/null` using a `printf`-family function. How can you recover what it's writing? For simplicity, assume that the process is writing continuously, in a loop.

Check yourself: download the silly program subliminal (compiled for 64-bit Illumos) from the course directory, and see if you can catch its output.

Extra credit: Make the outputs go to your terminal instead of `/dev/null`. I don't yet know how to do it gracefully.

Problem 8. *Hiders, keepers / Finders, weepers.*

In the following *extra-credit* problems, try to achieve the goal with the fewest possible modifications of the kernel state and the binary file, respectively.

- Hide an area of memory loaded into a process from `pmap`. Assume that the system administrator uses `pmap` to check which libraries are loaded into the process; you are allowed to have the data in some file in plain sight.

- Hide an ELF file section from casual inspection with tools such as `readelf` and `objdump`. Assume that a sysadmin who runs these tools knows the typical sections that go into an executable or shared object file.

- Find a way of hiding a file from `ls` run by root. I don't know a graceful answer to this on Illumos.