CS 258, Midterm Exam

Terms and Conditions. This is a take-home, open-book, open-manual, open-shell exam. The solutions are due by noon of Monday February 18.

You are expected to use (at least) DTrace, the OpenGrok source code browser[1] and the Modular Debugger's running kernel inspection capability[2], 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). 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.

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.

Problem 1. Enumerate the kernel-space data structures created (allocated and/or initialized) when a new process is created but before any of the code contained in the process' binary file is executed. Draw the relationships between them (you can sketch the data structures by hand or use the *Graphviz* (`http://www.graphviz.org/`) free software package.[3] You may limit your drawing to the ten data structures you consider the most important to describing a process.

Problem 2. When you start a kernel debugging session ("*mdb -k*"),

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

[2] `mdb -k`

[3] `http://www.graphviz.org/content/datastruct` is a good example if you want a neat graph; click on the image here and elsewhere in `http://www.graphviz.org/Gallery.php` to see the code that generates the picture.

MDB runs as a userland process. Yet it reads and interprets the kernel memory for you, which means that it has access to the information about the layout of kernel memory (i.e., kernel symbols).

- Where is this information stored and how does MDB access it?
- How does MDB access kernel memory?
- For a kernel global variable of your choice, show the appropriate data structures and their use by MDB.

(Previous years' lecture notes provide some hints, but your primary tools should be tracing and kernel state examination).

Problem 3. Write a C program that successfully modifies its own code (*.text* section) to break out of an infinite loop. In particular, consider the program with the following *main()* function:

```
int main(){
    while(1){
        puts("Woot!");
        breakout();
    }
    return(42); /* Hit this! */
}
```

Write the function *breakout()* that will break out of the infinite loop and hit the *return* statement by changing the program's own *.text* section.

You may use any system calls you need (except *exit()*). You may not use *goto*, *break*, and similar control flow statements.

You are also allowed to modify the compiled ELF executable of your program, but not the code inside it. Your solution will most likely depend on your compiler and platform; if so, turn in the *.s* file (the output of "*gcc -S* ") with it.

Hint: Use *truss* or *strace* to trace systems calls made by a process.

Problem 4. In an MDB session, enumerate all processes that share a physical memory frame that contains the *libc*'s *printf()* function, and all the virtual addresses at which this function is mapped into their respective address spaces. Bonus points if you manage to keep it to a one-liner of (chained) MDB or shell commands.

Problem 5. The file with Unix password hashes, */etc/shadow*, is not readable to processes run by regular users (only to *root*, and to processes started via *sudo* or *pfexec*). Start a user process that will attempt to read */etc/shadow*, and (with MDB and/or DTrace) modify its kernel data structures for the read to succeed. You can use "*cat /etc/shadow*" as a target process, or write your own. You may suspend the process and resume it as needed by means of DTrace or a debugger of your choice.