CSC 495 — Assignment 2 — Due Tuesday, March 3

1 CSC 495 — Assignment 2 — Due Tuesday, March 3

This is a one-week assignment, and will count for half as many points as the typical two-week assignment.

1. For this question you will perform a very simple exploit of a CWE-134: Uncontrolled Format String vulnerability. The goal here is a simple information-gathering exploit — there are more dangerous exploits that can take control of a system, but this question focuses on the more simple problem of finding a secret that is stored in memory.

   As a first step, study the C source code on `cmpunix` in `/csc495/hw2/q1/simple-fmt.c` — there is a pre-compiled executable saved as `simple-fmt` that you can run in that same directory. In the source code you should see a variable named `secretNum`, and your goal is to discover this value by running the program and providing the right input. This code has the simplest possible kind of format string vulnerability, and there are some additional helpful features to make it easier to exploit (notice the variables `clue1` and `clue2`). The executable is readable and was compiled with debugging information, so if you want to explore more than just looking at the source code you can run it in `gdb`.

   Your challenge: Once you have a good handle on how this program works and how to exploit the vulnerability, look in the `/csc495/hw2/q1/challenges` directory. There are "personalized" executables for each student in that directory, named by student user names. The only one of those that you can access is your own, and you have execute permission but not read permission (so you can't dump the program or run it in a debugger and find the secret that way). The source for these executables is exactly the same as `simple-fmt.c`, just with a different secret for each one. Exploit the format string vulnerability and find the secret (it's a number less than one million)!

   What to turn in: Describe what you did to exploit the vulnerability and find the secret. You should describe this in enough detail so that someone could read your description and repeat the steps to break the code. Next, describe *why* this worked (what precisely is going on when the exploit runs). Finally, give the secret number that you discovered.

2. For this question, you are going to explore a real program that has format string vulnerabilities in it. This is a "to-scale" problem — the source code is an `ftp` server that was widely used in the 1990s and early 2000s, and consists of a little over 22,000 lines of C code. While not large by today's standards (by comparison, the Apache web server has over 250,000 lines of code), this is certainly larger enough to make it infeasible to look for vulnerabilities without making use of tools to assist your search.

   In parts a-c below, you will need to figure out the correct shell command to solve the problem, mostly figuring out how to properly invoke `grep` and possibly pipe or redirect the output. You will need to turn in a "typescript" showing this command invocation. For full documentation on how to create a typescript, you can type "`man script`", but the only thing you really need to know is that typing

"`script ~/parta-script.txt`" will invoke a sub-shell and keep a record of your interactions with that shell session in the file named `parta-script.txt` in your home directory. It will keep recording everything until you exit the sub-shell, which you can do by typing the command `exit` at the shell prompt, or by simply pressing ctrl-d at the command prompt. Figure out the command you need to solve the problem first, and *then* start the typescript — I don't need to see all your failed attempts, just the one that works after you have figured it out! Each part below tells you what filename you should use for the typescript, so that I can find it. This problem depends heavily on you knowing how to use the `grep` command — if you haven't used this before, or aren't familiar with regular expressions, see the tutorials that are linked on the "MORE INFO" section of the class web page.

a. As a first step, gather some statistics on the source code, which is in directory `/csc495/hw2/q2` on `cmpunix`. Go into the `src` subdirectory and take a look. The source code is made up of `.c` files (C source code), `.h` files (C header files), and `.y` files (these are yacc files, which you may have never heard of, but don't worry about that!). How many of each kind of file are there? Don't just count by hand — use `ls` piped in to `wc` to find the answer.

   What to turn in: Leave a typescript named `parta-script.txt` in your home directory, as described above, showing your invocation of the correct command. In your written homework submission, describe how you found the answer (in other words, why the command shown in your typescript works) and give the actual answer (counts for various types of files).

b. Next, let's see how big the potential for format string vulnerabilities is. Format string vulnerabilities generally come from a call to either the `syslog` function or a function from the `printf` family (including `printf`, `fprintf`, `sprintf`, `vfprintf`, etc.). First, concentrate on `syslog`: come up with a `grep` invocation that will extract all the calls to the `syslog` function. Work on finding the right regular expression so that `grep` *only* extracts `syslog` function calls, and not things like `#include` directives for `syslog.h`. One you've figured out the right command there should be one line of output for each call to `syslog`, so pipe the output through `wc` to count the number of calls to `syslog`.

   Next, lets explore these calls to see if any have the potential for a format string vulnerability. The first idea is to capture the output of the `grep` command you just figured out so that you can go through the calls one-by-one to see if there are any problems. Unfortunately, some calls span multiple lines of source code, so `grep` doesn't extract the code you need to look at. Look up the `-A` option to `grep` in the man page and see what it does, and then experiment with including a `-A1` option to your `grep` command. Also, look up the `-n` option — use this so you can find the `syslog` calls easily later. Use these options and redirect the output of this `grep` command to a file so you have examine the `syslog` calls — note that you'll need to redirect to a file in your home directory (or somewhere similar) since you don't have write permission in the source code directory!

   Finally, go through the list and throw out calls that cannot cause format string vulnerabilities. Bring the `grep` output up in a text editor (`vi`, `vim`, `nano`, and `emacs` are all available on `cmpunix`). Go through and delete every `syslog` call in which the 2nd parameter starts with a quote — that's an instance where a literal string is used for the format string, so there's no chance of a format string vulnerability! For example, here's one set of 3 lines that resulted from my `grep` run:

```
ftpd.c:954:     syslog(LOG_ERR, "getpeername (%s): %m", argv[0]);
ftpd.c-955-#ifndef DEBUG
--
```

There's no way this can result in a format string vulnerability, so we don't need to consider it any more. Delete those three lines from your file of `syslog` calls! Keep doing this — once you get the hang of it, you can go through all of the `syslog` calls pretty quickly. Alternatively, you can copy-and-paste the potentially dangerous calls to a separate file. If you do this correctly, there will only be a very few potentially dangerous calls. How many did you find? Report the file and line number of each potentially dangerous call.

What to turn in: Leave a typescript named `partb-script.txt` in your home directory showing how you used `grep` to generate the initial list of function calls. In your written homework submission, describe how this command works and give the answers to the questions asked above (how many potentially dangerous system calls are there, and give a list by file and line number).

c. Repeat the last part for functions in the `printf` family. Find the total number of calls, the number of potentially dangerous calls, and list the file and line number of each potentially dangerous call. This is complicated by the fact that the format string can be at different positions in the parameter list (e.g., the first parameter for `printf`, the second parameter for `fprintf`, and the third parameter for `snprintf`), so make sure you are careful about this.

What to turn in: The same information as in part b, but save the typescript in your home directory as `partc-script.txt`.

d. For this part, you will take one of the potentially dangerous calls from part c and see if it really does cause a problem. Recall that a format string vulnerability arises when a user/attacker can provide the format string, so we will trace the source of each "potentially dangerous" format string back to see if it is in fact a problem. We will do this for the call to `vsnprintf` on line 5290 of ftpd.c, which you should have identified in part c.

First note that the format string is a variable `fmt` which comes in as a parameter to the function `vreply` — so where does this parameter come from? To determine this, find all calls of the function `vreply` and see what is provided for the 3rd parameter. There are only two choices: one in the `reply` function, and one in the `lreply` function. Finally: do the same thing you did in parts b and c for these two function calls. There are, in fact, a *lot* of calls to these two functions, so it's a lot to look through, but only a few are potentially dangerous. Do you see any that seem to be a particular problem? Explain in as much detail as you can what you discover in working through this part.

What to turn in: There's no need for a typescript for this part, so just answers the questions above in your written solution.

Note: Two of these calls were in fact the source of a real format string vulnerability that was exploited on real systems. Note that the vulnerability isn't obvious just from looking at `printf` calls — you have to trace those back to find where the arguments come from!