CSE 127: Computer Security (Spring 2019) **Due May 10, 10:00 P.M. PDT**

Assignment 3

100 pts

The goal of this assignment is to gain hands-on experience exploiting side channels. This assignment will demonstrate how side channels can be just as dangerous as the control flow vulnerabilities you exploited in Assignment 2. You will be provided a two skeleton files (`memhack.c` and `timehack.c`) that you will use to exploit side channels present in a target library (`sysapp.c`) to obtain a protected key. Your solution is due on May 10, 10:00 P.M. PDT. You may work with *one* other person in your class section on the assignment; if so you should only submit one solution for the two of you. You and your partner must *not* discuss your solution with other students until seven days after the assignment deadline. You may consult any online references you wish. If you use any code in your answer that you or your partner did not write yourselves, you *must* document that fact. Failure to do so will be considered a violation of the academic integrity policy.

1 Getting Started

To complete this assignment, you will be provided with a set of files including a turn-in script. All final testing should be done on cfbox vm provided with HW2, as that is where we will be evaluating your solutions. Please follow the instructions from HW2 regarding the vm setup and usage.

1.1 Assignment Files

Starter files are provided in an archive on the class webpage. It contains exploit starter code (`memhack.c` and `timehack.c`) which imports a library (`sysapp.c`) with password checking functionality vulnerable to side channel attacks. You should not modify `sysapp.c`, only `memhack.c` and `timehack.c`. The directory contains a Makefile to build your expoits.

1.2 Assignment Instructions

You will be writing two exploits, each of which takes advantage of a side channel, to obtain the password in `sysapp.c`. In `memhack.c`, you will exploit a memory-based side channel, and in `timehack.c` you will exploit a timing-based side channel. See Section 2 for specific details. Once both of your exploits can determine the password in `sysapp.c` and call the `hack_system()` function successfully, the assignment is complete. Additionally, for each exploit, provide a brief description of how it works in the `writeup.txt` file.

1.3 Submitting Your Solutions

To submit your solution, use the turn-in script provided with the starter files. It will archive your submission and submit it to our server to be graded. You may submit multiple times, but only your latest submission will be graded. Fill out the `PID` file with your PID on the first line. If you are working with a partner, include both PIDs on the first line separated by a space.

2 Exploit Construction

2.1 Memory-Based Side Channel

We recommend you start with the memory-based side channel because it is deterministic and doesnt have problems with noise. Look at the `check_pass()` function in `sysapp.c` and note two things:

1. The password string is passed by reference
2. The memory it points to is checked against the reference password one character at a time.

Now look in `memhack.c` and note how a buffer is allocated such that the page starting at `page_start` is protected (i.e., accessing it will caused a segmentation fault, or SEGV) and the previous 32 characters are allocated. Now look at the demonstration function `demonstrate_signals()`, which shows how referencing any memory in the protected page will produce a fault as well as how to catch that fault in your program. You do not need to use this function; it is merely there to show you how to use signals to capture whether a memory reference touched a page or not.

Now you will want to create a framework (in `memhack.c`) to call `check_pass()` with different inputs and catching any resulting faults so you can determine if part of the password is correct. We suggest a loop over the maximum password size (32 characters) where for the first guess you store the password such that its first character is one byte before `page_start` and then iterate between possible choices for the first character (when you get it right you will get a page fault). Repeat this to guess the entire password. Note that all ASCII symbols from ASCII 33 ("!") to ASCII 126 ("∼") can be used in the password. Other hints:

1. You are already given a page protected buffer with enough memory to crack the password. All you need to do is use it appropriately for each guess you make.
2. The `demonstrate_signals` function handles all the SEG faults for you. You can re-use almost all of it in your code.

2.2 Timing-Based Side Channel

Unlike the memory-based side channel, the timing-based side channel will deal with noise. Go back and look at `check_pass()`. An artificial delay has been added when each character is checked to make your life easier (it's possible to do the assignment without it but it would require a much more careful methodology). The execution time of `check_pass()` depends on how many characters you guess **correctly**.

Look in `timehack.c` and find a macro there for a function called `rdtsc()` which invokes the processors cycle counter (a clock that increments by one for each processor cycle that passes). In general, treat `rdtsc()` as a free running timer that returns a long. Insert a call to `rdtsc()` before the call to `check_pass()` and afterwards. Print the difference between these values to see how long (in cycles) the password checking ran. Run the program a few times. Now change the guess string so the first character is correct. Run again and see how the time difference changes.

Now automate this entire process, in the style of the original approach in `memhack.c`. Note that unlike the memhack attack, the timehack problem will have to deal with **noise**. Depending on things like what other programs are running, the status of the cache, the contents of the branch target buffer, etc... there can be significant variability in the amount of time each check takes. This **will** matter in practice. You will want to run a lot of trials before you reach your conclusion about each character. Other hints:

1. Be careful in using `printf`'s. These can blow out the instruction cache and data caches and perturb your results (i.e. overwhelm the timing effects you are trying to detect).
2. Be careful in averaging across trials. If your process is descheduled in the middle of a measurement, the time cost of that individual trial will be so large that it overwhelms everything else. Instead, the **median** is your friend. However, feel free to experiment if something does not work for you.
3. If time is not continuing to increase as you progress through characters, then you probably made a bad guess earlier. Backtrack.
4. `rdtsc()` will wrap around at some point. You may need to handle this outlier if it is causing issues.
5. Debugging advice: make a big array to hold your timing measurements and print them at the end.
6. Be sure to test a bunch of different passwords. We will when we grade.

2.3 Final Notes

Do not write a solution that simply checks all passwords exhaustively. You will not get credit for this. This should be doable in linear time (we will stop programs that are running for excessive periods) and it will basically feel instantaneous for passwords of 8 characters or less (note we will not test passwords over 12 characters). We plan to do the testing/evaluation on the cfbox vm.