UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Institute of IT-Security and Security Law

3rd IT Security Summer School Madagascar 2020

Cache-Based Side Channel Attacks

November 25, 2020

Prof. Hans P. Reiser, Johannes Köstler

Objectives

- Learn about cache-based side channel attacks.
- Implement simple Flush & Reload attacks.
- Implement simple Prime & Probe attacks.
- Visualize and analyze the results.

Resources

- C Programming Reference: `https://www.programiz.com/c-programming`
- Flush & Reload Attack: `https://eprint.iacr.org/2013/448`
- Practical Flush & Reload: `https://www.researchgate.net/publication/289952669_Cross-Tenant_Side-Channel_Attacks_in_PaaS_Clouds/`
- Practical Prime & Probe: `http://palms.ee.princeton.edu/system/files/SP_vfinal.pdf`

Report

- Explain the necessary steps.
- Document your source code.
- Include the measurement plots in your report.

INTRODUCTION

The introduction recaps some basic C programming, which forms the basis for the practical tasks in this project. Create a first C program (`hello.c`) with the following functionality:

1. Implement a `main` function that
   - stores a pointer to the first command line argument in a global variable `nameptr`
   - stores a pointer to dynamically allocated memory with `malloc()`, sufficiently large to store a copy of first command line argument, in a global variable `ucnameptr`
   - copies an upper-case version of the content referenced by `nameptr` to the dynamically allocated memory referenced by `ucnameptr`
   - calls a function `printhello(2)`(see below).

   Note: the use of global variables is, in many cases, bad programming style and most of the time should be avoided in real C programs.
2. Add a function `printhello(int n)` that
   - stores a reference to a constant string `"Hello "` in local variable `helloptr`
   - executes `n` iterations of a loop
   - in each iteration, prints a line containing the contents of the strings referenced by `helloptr` , `nameptr`, a colon (`":"`) and the string referenced by `ucnameptr`
3. Create a `Makefile` with targets

   all compiles and links everything, two separate steps for compiling and linking, producing executable file hello

   clean removes all automatically generated files
4. Extend the printhello function (at the end) with an additional instrumentation print statement that prints
   - addresses of content referenced by variables `ucnameptr`, `nameptr`, and `helloptr`
   - address of variables `nameptr` and `helloptr` in memory
   - address of functions `printhello` and `printf` in memory
5. In case your operating system is Linux you can investigate the memory layout of our program. Integrate code at at the end of the main function that:
   - extracts the addresses of variables `nameptr` and `helloptr` in memory
   - obtains the process identifier (PID) of the running process using the system call `getpid()` and reads the content of pseudo file `/proc/<pid>/maps` (`<pid>` is the return value of `getpid()`) by, for example, executing the external command "**`cat`**" with the `system()` library function).

```
char cmd[50];
snprintf(cmd, "cat /pro/%d/maps", getpid());
system(cmd);
```

For each of the addresses printed by the `printhello` function, find out to which memory section in the process map it corresponds to. Explain these results.

Exercise 1 Shared Libraries

In this exercise the program artifacts will be transformed into a shared library consisting of two parts: a shared library (`libprinthello.so`) that provides the implementation of the `printhello()` function, and the main program (`hello`) that invokes the shared library's `printhello()` function.

1. Split the source code into two C files (and appropriate header files) such that you can compile the `main()` part and the `printhello()` part independently (use static linking of the two parts to check if everything still works as before).

2. Transform the `printhello` part into a shared library. A shared library is not linked to your program when you generate the program's binary file. Instead, the operating system's dynamic linker links program and shared library when the program is executed.

   - You need to create position independent object code for the library, by invoking the compiler with the `-fpic` option
   - You can create the shared library from the object files by invoking
     `gcc -shared [object files] -o libprinthello.so`
   - You need to compile/link the main part such that it uses the shared library.
     - If you install your shared library in the system's default path, you can simply do so by linking the program with the `-printhello` option.
     - Otherwise, you need to tell the linker where to find the library using
       `-L <path-to-directory-with-library>`
     - If you (in the second case) run the generated binary, it will complain about not finding the shared library. You can fix this problem by telling the dynamic run-time linked in which directory to find the shared library. Two possibilities
       * Use `LD_LIBRARY_PATH` environment variable (see your favourite Internet search engine for an explanation)
       * Link the program with the option
         `-Wl,-rpath,<path-to-directory-with-library>`
   - Adjust your Makefile accordingly.

3. In which of the automatically generated files do you expect to find the constant string `"Hello "` defined in the source code? Can you verify this with simple tools?

4. Compare the output of the *address spaces* part with the output produced by the program from the introduction. Try to explain the differences.

Exercise 2 Experiments with RDTSC Time Measurements

Listing 1 shows the inline assembly code for measuring the access time of a specific memory address. The `reload` function takes a memory location pointer as input and returns the time to load the referenced data, whereas the `flush` function removes a memory chunk from the cache.

```
inline int reload(char *adrs) {
  volatile unsigned long time;

  asm __volatile__ (
    "  rdtscp             \n"
    "  movl %%eax, %%esi  \n"
    "  movl (%1), %%eax   \n"
    "  lfence             \n"
    "  rdtscp             \n"
    "  subl %%esi, %%eax  \n"
    : "=a" (time)
    : "b" (adrs)
    : "%esi", "%edx", "%ecx");

  return time;
}

inline void flush(char *adrs) {
  asm __volatile__ (
    "mfence"
    "\nclflush 0(%0)"
    : : "r" (adrs) :);
}
```

Listing 1: ”Memory Access Probe”

1. Describe the assembler instructions of Listing 1. Explain how the data is loaded and what time is measured.
2. Implement a simple test program that measures (many times in a loop) the access time for a single pointer (pointing to an address of the loaded shared library) and calculates the average value.
3. Implement a simple test program that measures (many times in a loop) the access time for the same pointer if you call the `flush` assembly inline function.
4. In addition to the average, also calculate the variance of the access time.
5. Record all measurements to a CSV file and create a graphical plot (X-Axis: iteration counter; Y-Axis: measured access time, use two different colors for ploting the measurements of both variants in a single graph).

Note: Plotting data in Python
An easy way to plot data is using the pyplot library. For example, if you have a CSV file with two lines (first line: X values, second line: Y values), you can plot it with this Python script:

```
import csv
import matplotlib.pyplot as plt
```

```
with open('test.csv', 'rb') as csvfile:
reader = csv.reader(csvfile, delimiter=',', quotechar='|')
x=reader.next()
y=reader.next()

plt.scatter(x,y,c=[1,1,0.2])
plt.show()
```

Listing 2: Plotting data with pyplot

Attack Preparations

You will find the code of the shared library (`libnumberpic`) and a sample implementation of two victim processes that use the shared library in an archive file `project02.tar.gz` at:
`https://itsec-madagascar.de/internal/`
It contains a `Makefile` for building both the shared library and two sample victim processes. The shared library renders numbers as png images and has the following API:

- `bitmap_t *numberpic_mkbitmap(int x, int y)`
- `void placenum(int nr, bitmap_t *bitmap, int x, int y)`
- `pixel_t *pixel_at(bitmap_t *bitmap, int x, int y)`
- `int save_png_to_FILE(bitmap_t *bitmap, FILE *f)`

The header file also defines the types `pixel_t` and `bitmap_t`. The function placenum can be used to draw a digit (0..9) at a specific position within a bitmap previously allocated with `numberpic_mkbitmap()`. The library contains an array of constant pixel arrays that define the visual representation of the digits (variable numbers).
The program **`test`** is a simple program that generates an image (**`test`**`.png`) containing the digits `1` and `4`.
The program `web` is a simple web server that takes two arguments: port number and files directory. If you run the program with `./web 8888 .` in the `src` folder, it accepts connections on port 8888 and serves files in the current directory (there is a sample `index.html`). On URLs addressing an image `ABCD.png`, it renders an image (png file) containing the digits `A`, `B`, `C`, `D`.
Implement, in a file `probe.h`, the inline functions for probing a memory location (measuring and returning the memory access time with `RDTSC`/`RDTSCP`) and for evicting a memory location from CPU cache (`CLFLUSH`).
You should be able to explain why you used exactly the instruction(s) you used, including possibly the use of some fence instructions.

- `inline __attribute__((always_inline))int reload(char *addr) { ... }`
- `inline __attribute__((always_inline))void flush(char *addr) { ... }`

Exercise 3 Basic Side Channel Probing

1. Checking access times

   In one console, run the first test program with `./test 1`.

   Write a simple probe program that – while the test program is running – periodically measures the access time of the memory pointed to by numbers [5] and numbers[7]. Make 1000 measurements, store them in a .csv file, and write a Python program that creates a graphical plot of the measurement results (X-Axis: iteration counter; Y-Axis: measured access time, use two different colors for ploting the measurements of [5] and [7] in a single graph).

   - Repeat the measurements on a different Laptop/PC if you have one available.
   - Repeat the measurements with and without CPU core pinning. You can use the `taskset` utility to enforce on which core the probe and the victim is running.

2. Basic side channel

   Write a simple probe program (`probe.c`) that detects if some process uses the `placenum` function of the shared library.

   - If no other processes access the shared library, it should not output anything.
   - If another process calls `placenum()`, it should print `Access` on `stdout` (if there is more than one call to `placenum()` within one second, it may print `Access` one or multiple times).
   - If another process calls other functions of the shared library, printing and not printing `Access` is an acceptable behaviour.

   You can test your program by starting probe and then concurrently

   - run `test 1` or `test 0`
   - run `web 8888 .` and point your browser to `http://localhost:8888/1357.png` (or some other number instead of `1357`). Press the reload button (or F5 key) in the browser...

Exercise 4 Advanced Side Channel Probing

1. Digit probing

   In this part, the attack target is the web server application `web`. On each request for png file `ABC.png`, it invokes `placenum()` for the digits `A`, `B`, `C`, ... to generate a picture shown on the web browser on the fly. The goal of the attacker is to find out

   - when a web client requests such a picture (solved already in the previous exercise)
   - what digits are shown in the image.

   Implement an extended attacker program `autoprobe.c` that continuously monitors the shared library's memory. It shall detect if a client accesses the web server to generate a png file with digits, and print which digits are shown in the image.

Client accesses `http://localhost:8888/1579.png`

Expected output `Access: [1], [5], [7], [9]`

The information must be retrieved using a cache side channel. `autoprobe` may not obtain the information by listening to network traffic, examining the web servers log file, or other means that do not use a cache side channel.

2. Enhanced information exfiltration from web form

   If you access `http://localhost:8888/index.html`, you find a web form in which you can type in a number, and after submitting the page (requires JavaScript) will show you a small animation with the number you typed in.

   Enhance the `autoprobe` program such that it takes a command line argument `t` (integer value in seconds).

   - If `t` is supplied, if monitors the shared library for `t` seconds, and tries to guess the correct number entered in the web form.
   - If `t` is not supplied, `autoprobe` shall maintain its normal behavior implemented in the previous exercise.