Lab 6 - Meltdown Attack

Due on Dec. 15^{th} (F) 2023 Midnight on Github

1 Overview

In this lab, you will implement 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. Build a mechanism for exception suppression.
3. Implement a Meltdown attack and steal the secret from our course workstation at *10.75.9.63 (Northeastern VPN required)*

2 Using the Course Workstation

1. You will receive an email to your husky email address with the subject "[EECE5699] Lab machine account ready," which contains the username and the password. If you could not find it, please check the spam.
2. Outgoing network traffic is disabled on that machine. To transfer files to it, please use sftp.
3. You are only allowed to run 40 processes at the same time.
4. Please write and debug your code on your local machine because the workstation has limited resources. Testing of your covert-channel should also be done on your machine. Only use the workstation for the "Stealing Secret" testing as the course workstation is using an obsolete Linux kernel that is still vulnerable to Meltdown.
5. Please back up your files to your local machine. The course workstation may be cleaned up at any time.

3 Accept your Assignment on Github

Please click the following link to accept this assignment: `https://classroom.github.com/a/zoeG6G5E`

4 Building a Covert-Channel (30 pts)

A covert-channel is built on top of a side-channel - shared microarchitecture. The sender encodes the data into the microarchitecture state, and the receiver decodes the data from the state. In this lab, you will build an in-process covert-channel using the Flush+Reload technique, where the sender and receiver are in the same process. An in-process covert-channel is much easier to implement than a cross-process covert channel and is sufficient for the 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 the F+R attack determines whether the victim has accessed a memory address or not (brought back to the cache). We now use the F+R technique in a slightly different way to build a 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 spaces (each at least of a cache line size). The covert-channel is done in three steps. First for presetting, a series of Flush operations flush all the 256 elements in the array one by one. Then for data transmission (encoding it in the cache), use the 8-bitt secret data as an index to access one of of the 256 memory locations. Last for receiving, the Reload operation (receiver) is to re-access the 256 memory locations one by one and time them. The memory location with a shorter access 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 4 via the cache covert-channel. A *UserArray* consists of 256 spaces where 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 - first flush 256 memory elements of *UserArray* out of caches.
2. Transmit data - the sender encodes the data into the cache state by accessing *UserArray[4 *space]*. With this access, the cache state changes to the 4^{th} memory block are 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 and time the reload. When you see a cache hit (lower timing), the i is the data value transmitted.

```
// define
space = 4096 // memory page size
UserArray = allocate (256 * space) bytes
data = 4

//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 on cache prefetching) and causing noise on the covert channel.

Module 1: for the covert channel you create, repeat with random bytes to transmit, and report the reliability of your covert channel. The reliability is defined by the ratio of successful decoding (data received) out of the total number of bytes transmitted. You can run this 100,000 times with random byte to transmit.

5 Exception Suppression (30 pts)

As the Meltdown attack is attempting to access a kernel address from user space, the user will get a segmentation fault (SIGSEGV) exception, and your program will terminate. You do not want this to happen as you would like to successfully steal the secret kernel data and transmit to a receiver. You need to catch the exception and resume the program using the C signal and setjmp library. A great tutorial on how to use the signal library can be found here (signal in C), and setjmp tutorial here (setjmp).

Here is an example of how to gracefully recover from a segmentation fault, which contains some helper functions for handling signals and exceptions.

```
1  #include <setjmp.h>
2  #include <signal.h>
3
4  jmp_buf buf;
5
6  //helper functions
7  static void unblock_signal(int signum __attribute__((__unused__))) {
8    sigset_t sigs;
9    sigemptyset(&sigs);
10   sigaddset(&sigs, signum);
11   sigprocmask(SIG_UNBLOCK, &sigs, NULL);
12 }
13
14 static void segfault_handler(int signum) {
15   (void)signum;
16   unblock_signal(SIGSEGV);
17   longjmp(buf, 1);
18 }
19
20 ...
21 //main function for recovering from a segmentation fault
22 int main(int argc, char** argv) {
23   if (signal(SIGSEGV, segfault_handler) == SIG_ERR) {
24     printf("Failed to setup signal handler\n");
25     return -1;
26   }
27   int nFault = 0;
28   for(i = 0; i < samples; i++){
29     test_illegal_address += 4096;
30
31     if (!setjmp(buf)){
32       // perform potentially invalid memory access
33       maccess(test_illegal_address);
34       continue;
35     }
36     nFault++;
37   }
38   printf("exception suppressed: %i\n", nFault);
39 }
40 ...
```

This is how the example works. In the main function, the initial call of *signal* function (at Line 23) links the *segfault_handler* function to handle the segmentation fault signal: SIGSEGV (faults generated when accessing memory incorrectly). Then in the following **for** loop (Line 28), 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 at Line 33. Suppose the test_illegal_address is an invalid address, you will get a segmentation fault, and a signal SIGSEGV is generated, which will direct the control flow to *segfault_handler* function. Inside the

segfault_handler function (Line 14), 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 31 will be evaluated to be false, and the control flow continues onto instructions after the *if* block, i.e., the segmentation exception is successfully suppressed. The program flow continues with Lin 36 and the loop (at Line 28). The tutorials mentioned above have much more detail about *setjmp* and *longjmp* functions.

Module 2: Write a program to trigger a segmentation fault on your local machine and suppress it. Provide a screenshot to demonstrate that.

The example below can trigger a segmentation fault (out of bound access for a[10000] which can be used as illegal memory access) :

```
1  int a[1] = {1};
2  a[10000]++;
```

6 Metldown - Stealing Kernel Secret (40 pts)

In this part of the lab, you will create the Meltdown attack and steal the secret at the kernel memory address placed in */tmp/target_address.txt* (e.g., 0xffff914f4c58ca80) on the class workstation. Note the timing threshold for the workstation is 150 cycles (this may be different from the timing threshold of your own machine, so you need to adjust the cache covert channel accordingly).

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 reading the kernel value instead of having a fixed value to transmit. The following is the code from Meltdown attack. This code implements Steps 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  );
```

On the course workstation, a victim is running to keep the kernel data warm in caches. The reason for doing so is on our machine, the out-of-order execution window is not longer than 36 CPU cycles, which is the time window that the transient kernel data tapping and encoding (transmission) have to happen. 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 to steal the secret. Use 'taskset -c 0 ./attacker' command to force the attacker program to run on core 0, where the victim is running on.

The message is a string starting from the given target address. Your code should access (tap) and transmit character by character. The string ends with '\0' which can be used as your program exit point.

Module 3: Putting everything together (the kernel data tapping, covert channel, and exception suppression) to implement Meltdown attack. Report the kernel message you have recovered.

7 What to Turn In

The submission includes all your code and a README file. The additional PDF report should include these items:

1. Report the reliability of your covert channel for Module 1.
2. A screenshot for Module 2.
3. Report your CPU model. e.g., Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
4. Report the kernel secret your retrieve from the class workstation for Module 3.