2162 Term Project 2

Spectre Attack

Assigned: 12/1/2021
Due: 12/16/2021

In this second project, your objective is to realize Spectre on a real computer.

1. **Spectre attack.**

To understand the Spectre attack, we need a little Operating System knowledge. The OS itself is just a special program that serves many purposes, but arguably the most important function is to share the CPU and other hardware (such as RAM) between multiple processes (programs).

For our purposes, the OS has two functions: schedule processes in some way (so that one process cannot just monopolize the CPU), and separate physical memory to provide isolation (so that process A cannot read/write process B's memory).

Scheduling: Our CPU has a timer which goes off every fixed interval (say 4ms), the OS runs, and chooses a new process to run. To save the old process, the OS dumps all the register values to a remembered location in memory. To load the new process, it restores the register values from another remembered location. The OS sets different locations for different processes.

Isolation: When a process is run for the first time, the OS gives a process a new piece of physical memory that is not being used by other processes (the OS keeps track of this information). Specifically, the OS sets up the page table for the new process. Process A and Process B are separated because their physical addresses are different. Each time a different process is scheduled by the OS, the OS configures the CPU to use the correct page table location (for virtual → physical address translation, as discussed in the Memory lecture slides).

While the memory is isolated, other hardware structures may not be, as security researchers are finding out. Specifically, the branch predicting hardware (branch predicting unit and BTB) are shared between processes.

What does this mean? If process A runs and uses a branch instruction, it influences the branch predictor state machine, as well as the BTB. Then, if the OS schedules process B and it uses a branch instruction too, the prediction is either TAKEN or NOT TAKEN, based on what process A did. Also, if the addresses of the branch instructions happen to line up correctly, the predicted **target** address for process B is influenced by the BTB state set by process A.

This is the core reason for the Spectre attack. Basically, an attacker constructs a program which trains the branch predictor as well as the BTB. Then, the OS schedules the victim, and once the victim process runs and uses a branch instruction, the predicted **target** address is used to load the next code into the pipeline. In other words, the attacker controls the prediction direction, as well as the predicted target, of the victim process.

Why is this a problem? If the attacker causes a correct prediction, then the victim just runs faster. If the attacker causes a wrong prediction, the CPU eventually just does a cleanup. The victim will take a performance penalty, but so what? Well, it turns out, that even though architectural state is flushed (registers are reset), and some microarchitectural state is reset (ROB, RAT), other microarchitectural state is not reset (such as cache).

This comes to the second part of understanding Spectre: **covert channels.** A covert channel is a channel that is not supposed to be used for communication purposes, but is used anyway by spies. **Many covert channels exist in modern CPUs**. This work is not new to the Spectre attack, but is needed to actually leak information from the victim process.

A CPU's cache can be used as a covert channel. Because the OS separates memory, two processes cannot (normally) communicate without special OS support. However, they can use cache if they are clever enough. If Spy A (using process A) and Spy B (using process B) share physical address X (e.g., through shared library), they can communicate in the following manner:
First, A flushes X from cache to memory at time t=0s. This can be achieved using clflush instruction on x86 processors.
Second, at time=1s, B can choose to either load address X or not.
Third, at time=2s, A loads address X. If X loads fast, it hits in cache; this means, at time t=1, B **did** load X into cache. If X_A loads slow, it missed in cache; this means, at time t=1, B **did not** load X into cache.

This enables B to transmit 1 bit of information every 2 seconds, with the following protocol: if B wants to transmit a 1, he loads X (causing X to load fast for A). If he wants to transmit a 0, he doesn't load X (causing X to load slowly for A). This covert channel is called Flush+Reload, which we will use in this project.

This enables one way communication. If they want two-way communication, they can either change the protocol, or choose another address Y. Amazingly, cache covet channels work very well that you can actually build applications over them; see this interesting paper[1]). Note that there are more cache covert channel than Flush+Reload, but until now, Flush+Reload is the most powerful one.

Now that you have a general idea on how to use a covert channel, we can finish the description of Spectre. In Spectre, Spy A is the attacker process, and Spy B is the **mispredicted code.** We don't need two-way communication: it's enough for the mispredicted code to send information to the attacker. The attacker looks at the victim's code (we assume this is available to the attacker), and finds a short piece of code which will leak a secret in cache. This is doable for a few reasons, but here's two: 1) In real CPUs, the misprediction window can be very long, so a lot of mispredicted code can be run 2) Attackers are good at finding code in applications which will leak secrets (**this code is called a gadget, which you need to know if you want to be a modern hacker).**

The short but sweet description of Spectre:

1) The attacker process mistrains the branch predictor and BTB and initializes the cache covert channel.
2) The victim runs, and mispredicts to a location in the victim's code which communicates a secret value via the cache side channel.
3) The attacker reads the cache side channel.

It's incredible that this really works in practice, but it does. Note that it's not necessary to use cache; other covert channels exist which can be used, it's just the cache is the most popular.

Now let's take a look at a code snippet of the victim in Spectre:

---

1 Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud

```
void victim_func(int x)
{
    if (x < array1_size)
    {
        int y = array2[array1[x]];
    }
}
```

When x is in bound, victim_func reads array1[x] and use its value as an index to read array2. When x is out of bounds, victim_func is not supposed to read any data from memory. However, if the branch prediction result is "taken'', array1[x] and array2[array1[x]] may be speculatively fetched from the memory hierarchy to the CPU core, before the CPU resolves the branch and knows that x is actually out of bounds and the branch should not be taken. These fetched data will eventually be dropped when the branch is resolved, that is to say, y will not be changed when x is out of bounds, for correctness. However, they may leave some hints in the hardware. For example, if array2[array1[x]] was stored in memory, after this speculative access, it is now brought into cache.

What if array1[x] (when x is out of bounds) is a secret value that is not supposed to be accessed by victim_func? As explained above, before the branch is resolved, we have already accessed the secret and brought it into the CPU core. But how do we leak it out? The answer is covert channel.

Since the secret is used as an index to fetch array2, we can flush all the entries in array2 from cache to memory before calling victim_func. Then, after executing victim_func, we reload all the entries in array2 and time the reload operations to determine which entry has been accessed (speculatively) and thus brought into cache. For example, if array2[5] is in cache (i.e., it is faster to load array2[5] than others). The secret value is 5. This is the Flush+Reload covert channel we explained earlier.

2. This project

[Project goal] Your goal in this project is to implement a Spectre proof of concept with Flush+Reload in a c program. To simplify the project, we set the victim and the attacker in the same program.

[Provided code] You are given three parts of the program (please find it in spectre.c).

1. The victim_func, which is essentially the one you see above.
2. Constant global variants array1, array2, and a secret string.
3. The function to flush an address to memory, and the function to load an address and time the load.

[Your part] Your job is to write the main function (the attack code) in which can print out the secret string without directly accessing it. Specifically, in the main function, you need to do three things:
First, flush array2 and mistrain the branch predictor.
Second, call victim_func and give a "correct'' x so that victim_func will speculatively load one byte of the secret string.
Third, reload all the entries in array2 and determine the secret value.
Fourth, repeat the above three steps to leak the entire secret string.

[Notes]

1. You are **not allowed to** modify victim_func and the global variants that are given.
2. Tricks for your attack to succeed:
   a. You need to figure out how to make it take a long time to resolve the branch. If the branch resolves too fast, there may not be enough time to load and encode the secret.
   b. Make sure your code, especially the part of training the branch predictor, is not optimized out by compiler.
   c. What is the timing boundry between a memory access and a cache access? You need to figure it out.
   d. Read the Spectre paper. https://spectreattack.com/spectre.pdf
3. System and Processor requirements:

a. You will need to do this on an Intel processor (which supports clflush instruction) with Linux system. You can figure out whether your processor support clflush by searching the processor name (Almost all of them do). Linux is required because the Flush+Reload code can only run with GCC compiler.
b. To compile the code, in a linux terminal, do "gcc spectre.c -O0"
c. If you already have a qualified processor, you can simply install a ubuntu virtual machine, following the instructions here: https://itsfoss.com/install-linux-in-virtualbox/
d. If you do not have access to any qualified processor, contact Dr. Yang, she can provide you access to her lab computers.

[To pass this project]
1) You must be able to write the attacker code, compile and run it.

2) Your attacker code must achieve the four steps mentioned in [your part].

3) You need to be able to demonstrate your code. During the demo, Dr. Yang may ask you to change the content of the secret string. You need to show her that your code is able to get the string value and print it out **without directly accessing it**. If you do something like "printf("%c ",secret_string[0])", it is a **failure**.